TA130 - Algoritmos y Programación

Notas de TA130

Sebastián Santisi

primer borrador[1]

2º cuatrimestre de 2024

[1] "Primer borrador": Este texto fue escrito en una pasada de punta a punta. No fue releído de forma crítica, no se verificó la sintaxis, gramática, ortografía, ni mucho menos los ejemplos en código, tampoco se incluyeron gráficos, ni tablas, ni resúmenes de capítulo, ni referencias a la biblioteca. Este texto es la primera estructura de una obra que se va a ampliar, completar y emprolijar.

Índice general

Capítulo 1

Introducción

El siguiente apunte intenta ser una guía de consulta complementaria a las clases de TA130.

1.1. Algoritmos y programación

Antes de entrar en detalles de implementación importa explicar el concepto de algoritmo.

Los algoritmos son una abstracción de pensamiento que existe desde milenios antes de que el ser humano piense ni siquiera en tener computadoras.

Un algoritmo es sencillamente una sucesión de pasos sistemáticos que sirven para resolver un problema. Por ejemplo, cuando en la escuela primaria aprendemos a multiplicar dos números decimales de n cifras lo hacemos utilizando un algoritmo determinado.

Los algoritmos tradicionalmente se explican de forma informal como una serie de pasos en lo que se conoce como un pseudocódigo. Insistimos en el punto de que son estructuras abstractas, así como nuestro lenguaje es abstracto. Somos nosotros los que tenemos la capacidad de interpretar de esa explicación los pasos a seguir.

Por el otro lado, los programas son implementaciones de algoritmos para ser ejecutados por una computadora. Ahí ya no hay abstracción, las computadoras necesitan una secuencia de instrucciones estrictas para ser ejecutadas en un determinado orden, sin ningún tipo de ambigüedades.

A diferencia de los algoritmos, que se expresan en lenguaje natural, los programas se implementan en en lenguajes de programación. Los lenguajes de programación tienen reglas estrictas de sintaxis sin ambigüedades.

Volviendo a los algoritmos y la programación, ambas cosas son independientes entre sí y en este curso nos centraremos en ambas por separado. Necesitaremos aprender un lenguaje para programar nuestros algoritmos, pero además necesitaremos herramientas más abstractas para diseñar nuestros algoritmos previo a programarlos. Por lo general no hay un único algoritmo para realizar una misma operación y puede haber mucha diferencia en el rendimiento (por ejemplo medido en cantidad de operaciones o de memoria) entre diferentes variantes, cosa que no tiene nada que ver con los detalles de implementación en un lenguaje en particular. Retomando el ejemplo de la escuela primaria aprendemos a multiplicar dos números de n cifras con un algoritmo que realiza $n \times n = n^2$ multiplicaciones de 1 cifra, sin embargo existen algoritmos como por ejemplo el algoritmo de Karatsuba que utiliza menos de $3 \cdot n^{1,58}$ multiplicaciones.

1.2. Procesadores y programación

Las computadoras son dispositivos que tienen la característica de ser programables. La circuitería de base de la computadora se llama hardware, mientras que el programa que se carga sobre ella se llama software. En esta materia vamos a centrarnos en la construcción del software.

Si bien estamos hablando de computadoras en el ámbito de la electrónica hay muchos dispositivos que contienen un microcontrolador programable y no necesariamente tengan el aspecto de lo que usualmente interpretamos como una "computadora" (un monitor, un teclado, mouse, gabinete, etc.). En este curso nos centraremos en programar para computadoras, pero el área de aplicación de la electrónica también incluye a los dispositivos "embebidos" que contienen microprocesadores en su interior.

Genéricamente un dispositivo programable consiste de dos áreas: Un procesador y una memoria.

Una memoria es un dispositivo con determinada capacidad que permite almacenar datos (usualmente bytes) en una determinada posición numerada para recuperarlos después. En la memoria podemos encontrarnos valores que queremos recordar, pero también va a ser donde se aloje el programa que queramos ejecutar.

El procesador es un dispositivo que sabe cómo ejecutar un determinado número de operaciones. Cada operación es una acción atómica del estilo de sumar dos datos, traer un dato de la memoria, almacenar un resultado en ella, tomar una decisión, etc. Las operaciones se ingresan en forma de instrucciones que no son más que un número que codifica el código de operación y los parámetros de la misma. Esto es lo que constituye el código de máquina, y los programas no son otra cosa que una sucesión de instrucciones. Cuando el procesador ejecuta un programa no hace otra cosa que ir decodificando de forma secuencial las instrucciones de un programa y ejecutando las operaciones que allí se contienen.

Dado que las instrucciones de código máquina son valores binarios difíciles de memorizar lo más usual es que si se quiere generar un programa a partir de sus instrucciones las mismas se codifiquen en lenguaje de ensamblador. El lenguaje de ensamblador (assembly) consiste en ponerle nombres amigables a cada una de las operaciones, estos nombres se llaman mnemónicos. Entonces por ejemplo una operación en vez de ser la operación 13 se transformará en la operación `ADD` (suma en inglés), y por ejemplo la operación de sumar el dato del registro B que se identifica con el 2, en vez de ser una codificación de ese 13 con el 2 que prodría dar algo estrafalario como 210, se programará como `ADD B`. Escribir un programa que codifique instrucciones de assembly en código máquina es muy sencillo y desde el inicio de la programación que se utilizan de estos programas, llamados ensambladores, para poder programar en assembly y generar con él los programas.

Como cada procesador tiene sus propias operaciones el código de máquina es único para cada modelo de procesador. Es decir, los programas codificados para un determinado procesador no funcionarán en otro (y también el assembly de cada procesador será único). En este contexto es donde surgen los lenguajes de programación. La idea de los lenguajes de programación es poder expresar las operaciones de forma semántica y dejar que otro pograma, el compilador, decida qué operaciones de assembly hay que utilizar en determinada plataforma en particular para concretar esa operación. Los lenguajes de programación no están atados a una plataforma en particular y, siempre y cuando haya un compilador disponible, podremos compilar nuestro programa para ejecutarse en un procesador determinado.

1.3. El lenguaje de programación C

Los primeros lenguajes de programación surgieron en la década de 1950, y hay un sinnúmero de ellos disponibles. En este curso utilizaremos particularmente el lenguaje C.

La historia de C nace de la mano del sistema operativo Unix. En el año 1970 Ken Thompson y Dennis Ritchie, entre otros, desarrollan este sistema operativo para la computadora DEC PDP-7. El código de Unix se migra posteriormente a la computadora siguiente PDP-11/20 y queda en claro que desarrollar un sistema operativo en assembly no es eficiente porque hay que reescribirlo por cada nuevo procesador.

En ese momento había una gran variedad de lenguajes de programación orientados a las matemáticas, a los sistemas bancarios, inteligencia artificial, enseñanza, etc. pero no había ningún lenguaje de programación orientado a la escritura de sistemas operativos. Para escribir un sistema operativo lo que se necesita es un lenguaje con operaciones de suficiente bajo nivel para poder ser traducidas casi uno a uno en instrucciones de assembly, sin abstracciones de alto nivel que sean complejas de traducir a lenguaje de máquina. Además se necesita que el lenguaje provea de una interfaz simple para acceder a los recursos de hardware. Con esas premisas es que en el año 1972 Dennis Ritchie desarrolla el lenguaje de programación C, lenguaje en el cual se reescribe completo Unix después.

Hoy en día, más de 50 años más tarde, C sigue siendo uno de los pocos lenguajes de programación que permite acceder de forma transparente al hardware que está debajo y que conjuga la elegancia de un lenguaje de programación con la potencia del assembly, pero garantizando la portabilidad a cualquier procesador.

Dicho sea de paso, que el lenguaje de programación C date del año 1972 no significa que hoy en día se programe el C de ese año. El lenguaje ha sido actualizado múltiples veces, y ha pasado por un proceso de estandarización con estándares ISO que son revisados y mejorados cada una decena de años. El primer estándar data del año 1989, pero hay estándares 1999, 2011, 2018 y actualmente se está elaborando un estándar nuevo. En este curso en particular nos centraremos en el estándar ISO-C99.

1.4. El hola mundo

En el año 1978 Brian Kernighan y Dennis Ritchie publican el libro El Lenguaje de Programación C, conocido simplemente como K&R (Kernighan and Ritchie) y en él introducen el lenguaje a partir de un programa que saluda al usuario. La influencia de ese libro ha hecho que desde ese entonces el "hola mundo" sea el estándar para presentar cualquier lenguaje de programación.

El hola mundo en C tiene el siguiente aspecto

hola.c

```c
#include <stdio.h>

int main() {
    printf("Hola␣mundo\n");
    return 0;
}
```

y aunque en principio es el programa más sencillo que podamos hacer, explicar cada una de sus líneas implica adentrarse en cómo funciona el lenguaje. Dejémoslo para un poco más adelante.

Lo que mostramos recién es el código fuente del hola mundo, ahora bien, eso no es un programa, es decir, no es un conjunto de instrucciones que pueda ejecutar un procesador. Es en realidad la receta para que un compilador pueda construir ese programa.

Si quisiéramos construir un programa deberíamos meter el contenido del hola mundo en un editor de textos sin formato y guardarlo como un archivo, por ejemplo `hola.c`, para luego compilarlo.

El compilador que vamos a utilizar en este curso se llama GCC (GNU compiler collection) y vamos a asumir que lo tenemos instalado en nuestro sistema GNU/Linux y que accedimos a una terminal del sistema.

Si quisiéramos compilar `hola.c` para generar el ejecutable `hola.exe` en la terminal escribiríamos lo siguiente:

```
$ gcc hola.c -o hola.exe
$
```

Si todo estuviera bien el compilador generaría el programa `hola.exe` y no impriría salida, en cambio si hubiera errores el compilador nos indicaría de qué errores se trata.

Una vez generado el `hola.exe` podríamos ejecutarlo de la siguiente manera:

```
$ ./hola.exe
Hola mundo
$
```

Esa es la ejecución de nuestro primer programa.

1.5. Funciones

Presentemos el siguiente código:

```
1 int cuad(int n) {
2     return n * n;
3 }
```

Ese código define una función llamada `cuad()`. Una función en programación es asimilable a una función en matemática[1]. Desde afuera podemos pensarla como una cajita negra en la que entran valores y la función nos devuelve otros valores.

Esta función recibe como parámetro un valor llamado `n` de tipo entero (**`int`**) y devuelve a su vez un valor entero. La estructura de declaración de una función es:

```
1 tiporetorno nombrefuncion(tipoparametro1 nombreparametro1,
    ↪ tipoparametro2 nombreparametro2, ...);
```

Internamente la función devuelve (**`return`**) el resultado de computar `n * n`, es decir la multiplicación de `n` por si mismo, o sea `n` elevado al cuadrado[2].

Las funciones nos van a servir para encapsular operaciones complejas, cosas que queramos reutilizar, para jerarquizar complejidades en nuestros programas, entre otras cosas.

Una función por sí sola no hace nada, una función es un fragmento de código que se ejecuta únicamente si la función es invocada. Invoquemos a la función `cuad()`:

```
1 #include <stdio.h>
2
3 int cuad(int n) {
4     return n * n;
5 }
6
7 int main() {
8     int cuad_dos = cuad(2);
9     printf("El␣cuadrado␣de␣2␣es␣%d\n", cuad_dos);
```

[1] Ojo, que sean similares no quiere decir que sean lo mismo.

[2] De ahí el nombre de la función.

```
10      return 0;
11  }
```

Hay muchas cosas para explicar, vayamos por partes.

El fragmento que dice `cuad(2)` es una invocación a la función `cuad()`. Internamente el programa interrumpe su flujo de ejecución para ejecutar el código de la función `cuad()`. Como estamos invocando a la función con el valor `2` ese va a ser el valor que va a tomar `n` adentro de la función. Internamente se va a operar `n * n` que, como `n` vale `2`, va a dar como resultado `4` y finalmente la función va a devolver ese valor `4`. Es decir, después de llamar a la función se va a continuar con el flujo y donde decía `cuad(2)` el resultado de esa invocación será `4`.

La expresión `int cuad_dos` declara una variable llamada `cuad_dos` de tipo entero. Las variables sirven para almacenar valores que queremos recuperar después. El operador `=` es la asignación, `a = b` le asigna a la variable `a` el resultado de la expresión `b`. En este caso, almacenamos el `4` en la variable `cuad_dos`.

Cuando hicimos el hola mundo no lo sabíamos pero ahora sí: `printf()` también es una función y la estamos invocando. Esta función lo que hace es imprimir por la terminal lo que le pidamos. Prestar atención a que esto se aparta por completo del concepto de funciones que tenemos en la matemática, no invocamos a `printf()` para que nos devuelva algo[3] sino que la invocamos para que haga algo por fuera del mecanismo de pasaje de parámetros, devolución de valores.

`printf()` es una función compleja que puede recibir un número variable de parámetros, ahora bien el parámetro más importante es el primero, que es la "cadena de formato". Esta cadena le dice a `printf()` qué es exactamente lo que tiene que imprimir, y además le indica qué parámetros adicionales se han utilizado. En este caso el modificador `"%d"` le dice a `printf` ↪ `()` que espere un parámetro adicional de tipo entero. Al ejecutar la función imprimirá por la terminal `"El␣cuadrado␣de␣2␣es␣4\n"`, es decir, el `"%d"` se reemplazará por el valor de la variable `cuad_dos`.

¿Qué representa el `"\n"` al final de la cadena de formato de `printf()`? Probá sacándolo y fijate qué pasa.

Ahora bien, ¿de dónde salió `printf()`, quién la programó, cómo es que puedo usarla si no la definí? Bueno, `printf()` es una función de la *biblioteca* de C. La biblioteca de C es provista por el compilador y trae un montón de funciones auxiliares que no necesitamos programar[4]. Vamos a entrar en detalles más adelante.

Llegado a este punto podemos explicar qué significa la línea `#include <stdio.h>` que está al comienzo. Esa línea *incluye* las cosas necesarias para que el compilador "sepa" cómo es `printf()`. La palabra `stdio` significa standard input/output, o sea, entrada/salida estándar, y declara todas las funciones que tienen que ver con leer y escribir datos.

Nos falta explicar una última cosa y es el elefante en la habitación: `main()` también es una función. Particularmente se trata de una función que no recibe ningún parámetro y que devuelve un valor entero. La función `main()` es una función especial. Es el "punto de entrada", es decir la función que se ejecuta cuando se inicia nuestro programa. Toda la funcionalidad que implementemos va a estar comandada desde esta función, lo cual no quiere decir que no podamos llamar a otras funciones, como efectivamente estamos haciendo en el ejemplo.

El valor de retorno de `main()` es un retorno hacia afuera del programa, al sistema operativo. La convención es que si nuestro programa terminó correctamente devolvamos `0`. A estas alturas del conocimiento de C todos nuestros programas van a terminar correctamente así que de momento devolveremos siempre `0`.

[3] Lo cual no quita que `printf()` nos devuelva algo, que como no nos importa no lo estamos capturando en ninguna variable.

[4] Y que en algunos casos tampoco sabríamos cómo programar, porque para desarrollar la funcionalidad de `printf()` necesitaríamos conocer un montón de detalles sobre la plataforma donde estamos compilando y la gracia de utilizar un lenguaje de programación era justamente la de abstraerse de la plataforma y dejarle ese dolor de cabeza al desarrollador del compilador.

Y con esto no sólo explicamos el ejemplo, explicamos además todas las cosas que están en el hola mundo.

Capítulo 2

Sintaxis básica de C

Como ya dijimos los lenguajes de programación, a diferencia de los idiomas que hablamos, tiene una sintaxis estricta que no deja lugar para ambigüedades. Entendemos por sintaxis la estructura del lenguaje, cómo se forman sus "oraciones", cuáles son sus conectores, etc. En paralelo a ello tenemos la semántica que es la intención de una expresión. Una expresión puede ser sintácticamente correcta, pero no tener ningún sentido desde la semántica.

En las siguientes secciones iremos abordando diferentes partes de la sintaxis básica del lenguaje C.

2.1. Identificadores

Los nombres de las variables y funciones, llamados identificadores, pueden contener letras de la `a` a la `z`, de la `A` a la `Z`, números del `0` al `9` y guiones bajos (`_`) con la salvedad de que no pueden empezar con un número.

Por ejemplo, son identificadores válidos `suma_cuadrados`, `sumaCuadrados`, `SumaCuadrados`, `_suma_cuadrados`, `SUMA_CUADRADOS` o `SumaCuadra2`, mientras que son inválidos `2SumaCuadra`, `SumaCuadra-2` o **`int`**.

¡Momento!... ¿por qué el último es inválido, si bien cumple las reglas? Es inválido porque **`int`** es una palabra reservada del lenguaje. Son palabras reservadas todas aquellas que ya significan algo diferente en C. La siguiente es la lista completa de palabras reservadas del lenguaje:

```
auto double int struct break else long switch case enum register typedef char
↪ extern return union const float short unsigned continue for signed void default
↪ goto sizeof volatile do if static while _Bool _Imaginary restrict _Complex inline
↪
```

2.2. Expresiones

Las expresiones en C son las construcciones que al ser evaluadas resultan en un valor. Las hay de diferentes tipos:

Literales: Los literales son las expresiones que *literalmente* ya representan un valor en sí. Sencillamente evalúan a ese valor. Ejemplos: Números enteros: `0`, `-3`, `42`, números de punto flotante: `0.0`, `-3.3`, `42.92039`, `6.022e23`, cadenas de caracteres: `"Hola␣mundo"`.

Variables: Las variables evalúan al valor almacenado en dicha variable. Por ejemplo: Si hemos definido **`int`** `i = 5`, si luego operamos `i + 1` la expresión `i` evaluará a `5`, el valor que almacenó.

Operaciones: Las operaciones permiten combinar otras expresiones y evalúan a lo que compute esa operación. Por ejemplo `5 + 7`, `5` y `7` son literales y evalúan a su valor y la operación `+` es la suma, por lo tanto la expresión evaluará a `12`. Dado que las operaciones permiten combinar expresiones y a su vez las operaciones son expresiones esto significa que se pueden hacer expresiones con operaciones tan complejas como uno quiera, por ejemplo `(a + 5)/ 3 + b`.

Llamadas a función: Las llamadas a función evalúan a lo que devuelva la función para las expresiones que reciba como parámetros. Por ejemplo `cuad(1 + 2)` evaluará a `9`.

Asignación: Si bien en C la asignación `=` es un operador, es decir su comportamiento está englobado en el ítem "Operaciones" vale la pena explicarlo por separado. La operación de asignación **devuelve** el valor que asignó. ¿El operador de asignación no servía para asignar? Sí, también, además modifica el valor de la variable que esté a izquierda, pero el operador de asignación como operador devuelve el valor de lo que asignó. Por ejemplo, la expresión: `a = b = 5` se ejecuta con la siguiente asociatividad `a = (b = 5)`, es decir, primero asigna `5` a `b`, ahora bien, ¿qué asigna en `a`? Bueno, como se dijo, el operador devuelve lo que asignó, es decir `5`, por lo que en `a` también se asignará `5`.

Pregunta: ¿Cuánto valen `a` y `b` después de ejecutar la siguiente expresión: `b = 5 + (a =` ↪ `cuad(3 - 1)+ 2);`

2.3. Operadores

Los operadores de C pertenecen a varias categorías, en esta sección vamos a ver algunas de ellas.

2.3.1. Aritméticos

+: Operador de suma. Ej.: `2 + 2`, evaluación `4`.

-: Operador de resta. Ej.: `100 - 1`, evaluación `99`.

***:** Operador de multiplicación. Ej.: `2 * 5`, evaluación `10`.

/: Operador de división. Ej.: `7 / 2`, evaluación `3`.

%: Operador del resto de la división. Ej.: `7 % 2`, evaluación `1`.

2.3.2. Signo

-: Signo negativo. Ej.: Si `a = 5`, `-a` evalúa a `-5`.

+: Signo positivo. Ej.: Si `a = 5`, `+a` evalúa a `5`. (Sí, es muy útil...)

Notar que si bien el símbolo de estos operadores es el mismo que el de los aritméticos la diferencia es la aridad de los mismos. Los operadores aritméticos son binarios, es decir, necesitan dos valores para operar, mientras que estos operadores son unarios (o monarios).

2.3.3. Asignación:

En principio el operador de asignación es el = y como ya se dijo, además de modificar el valor de la variable a izquierda, devuelve el valor que asignó.

El operador de asignación se puede combinar con los aritméticos para *actualizar* el valor de una varible.

Ejemplo, si `a = 5` la expresión `a += 2` es equivalente a `a = a + 2` por lo que luego de ejecutarse `a` valdrá `7`.

De forma análoga esto mismo funciona para los operadores `-=`, `*=`, `/=`, `%=` (en estos últimos dos prestar atención a que la actualización es sobre el numerador).

Las operaciones de actualización son algo muy común en la programación y preferiremos utilizar estos operadores por una cuestión de claridad en la intención (semántica) de una expresión.

2.3.4. Incrementos:

Entre todas las operaciones de actualización, la más común es la de incrementar en uno o decrementar en uno una variable y para ello el lenguaje C provee no uno si no 4 operadores.

El operador `++` incrementa en uno el valor de una variable, mientras que el operador `--` decrementa en uno el valor de una variable. Ahora bien, hay dos variantes de cada uno y difieren no en cómo modifican a la variable si no en su valor de evaluación dentro de una expresión.

Cuando el operador `++` (`--`) se encuentra a **izquierda** de la variable es el operador de preincremento (decremento) y el mismo evalúa al valor **nuevo** de la variable.

Ejemplo:

```
1 int b, a = 5;
2 b = ++a;
```

`a` vale `6` y también vale eso `b`, porque el resultado evaluó al valor de `a` luego de ser incrementado.

Cuando el operador `++` (`--`) se encuentra a **derecha** de la variable es el operador de postincremento (postdecremento) y el mismo evalúa al valor **viejo** de la variable.

Por ejemplo:

```
1 int b, a = 5;
2 b = a++;
```

`a` vale `6`, porque la incrementamos, pero en este caso `b` vale `5` porque el operador evaluó al valor de `a` previo al incremento.

Dicho sea de paso, notar que `++a` es una expresión totalmente diferente a `a + 1`. Si bien ambas evalúan al valor que da incrementar `a` en uno, la primera modifica a la variable mientras que la segunda no. Es más, la primera **requiere** obligatoriamente que haya una variable mientras que la segunda puede ser una operación entre expresiones literales, o de llamada a función o una expresión derivada.

2.4. Precedencia y asociatividad.

Todos los operadores tienen una precedencia definida. Para simplificar diremos que (en la mayor parte de los casos) las mismas son las que esperaríamos que tuvieran.

Es decir, si evaluamos `2 * 3 + 1` la multiplicación tendrá precedencia sobre la suma y el resultado será `7`.

Si quisiéramos forzar una precedencia diferente de la predeterminada podemos utilizar paréntesis: `2 * (3 + 1)` dará como resultado `8`.

El operador de asignación tiene la más baja de las prioridades, o sea, es el último que se ejecuta.

En cuanto a la asociatividad, la misma en C es de izquierda a derecha. Es decir, en la expresión `1 + 2 + 3` la misma se ejecutará como si fuera `(1 + 2)+ 3`. Ahora bien C **no** define un orden de ejecución, las expresiones pueden evaluarse en cualquier orden. En el ejemplo dado que se trata de literales esto carece de importancia, pero si la expresión fuera `f() + g() + h()` por más que la asociación sea de izquierda a derecha no hay garantía de en qué orden se van a evaluar las funciones, y si las mismas modificaran algún estado interno o interactuaran con el exterior el resultado sería indefinido[1].

Todas las asociatividades son de izquierda a derecha con excepción de las asignaciones, por ejemplo `a = b = 5;` evalúa de derecha a izquierda: `a = (b = 5);`.

2.5. Instrucciones

Un programa se compone de instrucciones o sentencias. Serían las "oraciones" de nuestro código, cada una con un significado independiente de las demás.

Las instrucciones se separan por `;` y generalmente se escribe una instrucción por línea, si bien al compilador no le importa cómo acomodemos el código.

En el ejemplo del hola mundo, la línea del `printf()` representa una función así como lo representa la línea del **`return`**.

Las instrucciones pueden constituirse en bloques, los bloques se definen en C con un `{` para iniciarlo y un `}` para cerrarlo. Los bloques se pueden anidar, por lo que cada `}` termina el último bloque que se abrió.

En los ejemplos vistos hasta el momento utilizamos bloques para definir el contenido de las funciones `main()` y `cuad()`.

Generalmente una instrucción se constituye por una expresión de las que ya desarrollamos, pero también son instrucciones la declaración de una variable, la declaración de una función, el **`return`** y las estructuras de control de flujo que veremos más adelante.

2.6. Declaración de variables

En el lenguaje de programación C una variable tiene que ser *declarada* antes de ser utilizada.

Para declarar una variable, o un conjunto de variables se escribe primero el tipo de la variable y luego el nombre (o los nombres) de la misma.

Por ejemplo:

```
1 int a;
2 int b, c;
```

Además una variable puede o no ser *definida* en el momento de su declaración. Definir una variable es darle un valor. Por ejemplo:

```
1 int a, b = 5;
```

En este caso estamos declarando la variable `a` pero declarando **y** definiendo la variable `b` con un valor de `5`.

[1] Cuando decimos que un resultado es "indefinido" no significa que sea un resultado ilógico, en este caso el resultado tiene que ser alguna de las combinaciones de evaluaciones de las funciones. Indefinido significa que no sabemos cuál y por lo tanto nuestro programa puede hacer cosas diferentes según decida el compilador. Esto implica que nuestro código es ambiguo, lo cual viola la idea de que un programa no debería serlo. Si quisiéramos eliminar la indefinición, podríamos tranquilamente evaluar las funciones previamente en el orden que queramos, guardar los resultados en variables y luego operar con los resultados.

Siempre que utilizamos una variable en una expresión (es decir, que queramos evaluar su valor) tiene que haber sido definida previamente. En el lenguaje C si una variable no hubiera sido definida su contenido es indefinido y dará lugar a resultados impredecibles.

2.7. Declaración de funciones

En los ejemplos vimos cómo *definir* una función con la secuencia tipo de retorno, nombre de función, parámetros y entre llaves el bloque de instrucciones que constituye su implementación:

```
1 int sumar(int a, int b) {
2     return c;
3 }
```

La primera línea de la definición representa la firma, prototipo o interfaz de esa función. Es la que dice cuáles son los parámetros formales de la función.

Ahora bien, hay veces en las cuales sólo queremos *declarar* una función para darle a conocer al compilador la firma de una función pero no proveer (todavía) la definición de la misma. Para ello escribimos la firma seguida de un `;`:

```
1 int sumar(int a, int b);
```

Esto no nos libera de proveer una implementación de la función `sumar()` eventualmente, pero dado que el compilador trabaja de arriba hacia abajo en una única pasada, permite poder utilizar funciones que aún no están definidas.

Hay dos usos primarios para declarar una función. El primero es un tema de estructura de código, si cada función tuviera que estar definida antes de ser utilizada nos obligaría a tener todas las funciones ordenadas según cuál usa a cada cuál e incluso no podría resolver casos donde dos funciones se llamaran mutuamente. La declaración resuelve este problema y permite estructurar libremente el código.

El otro uso de la declaración de una función es cuando la implementación va a ser provista por otro mecanismo. Ese es el caso de, por ejemplo, el uso de la función `printf()` que utilizamos pero no implementamos. Y ahora sí terminamos de explicar el ejemplo del hola mundo, la instrucción `#include <stdio.h>` de la primera línea incluye en mi código fuente el archivo de *encabezados* (headers, de ahí `.h`) `stdio.h` provisto por el compilador. En él no está la definición de la función `printf()` pero sí su declaración, lo cual permite que yo pueda utilizarla. La implementación de la misma se incorporará a mi ejecutable en un paso posterior a la compilación que explicaremos más adelante.

2.8. Comentarios

Muchas veces hay porciones de código que necesitan documentación de qué representan. Esta documentación no forma parte del código que le importa al compilador si no que le importa a otros programadores. Para esto se utilizan los comentarios.

C tiene dos variantes de comentarios, de múltiples líneas o de final de línea. Los comentarios de líneas múltiples inician con un `/*` y se terminan con un `*/`, mientras que los de final de línea van desde un `//` hasta el final de la línea. Por ejemplo:

```
1 /*
2   Calcula el módulo de un vector 2D.
3    Argumentos:
4       float x: coordenada de las abscisas
5       float y: coordenada de las ordenadas
```

```
6     Devuelve: (float) el módulo del vector
7  */
8  float modulo_vector(float x, float y) {
9      return sqrt(x * x + y * y); // sqrt -> raiz cuadrada, incluir
           ↪ <math.h>
10 }
```

Capítulo 3

Datos

Cuando en el primer capítulo definimos al procesador dijimos que el mismo consistía en un procesador y una memoria, donde la memoria almacenaba bytes accesibles de forma individual sabiendo su posición.

Queremos remarcar eso: la memoria guarda paquetitos de ocho unos y ceros y desconoce qué representan esos valores. Desde el punto de vista de la memoria en determinada posición hay un determinado patrón de bits que representa un byte.

Dado que la computadora opera con ceros y unos se puede decir que la computadora guarda valores binarios.

Los seres humanos solemos usar el sistema decimal para la mayor parte de las cosas y lamentablemente no hay una manera directa de traducir decimal a binario y viceversa sin hacer una secuencia de operaciones.

En el lenguaje C los literales pueden ingresarse en decimal, como ya vimos hasta el momento, pero también en base octal y en base hexadecimal:

```
1 1234    // Valor decimal
2 01234   // Valor octal
3 0x1234  // Valor hexadecimal
```

Son octales los literales con un número impar de ceros a izquierda, y son hexadecimales los literales con un `0x` a izquierda.

Internamente los 3 números ingresados se guardarán según su representación binaria e, irónicamente, no hay manera de ingresar literales binarios en el estándar C99.

Las bases 8 y 16 son de particular interés dado que $2^3 = 8$ y $2^4 = 16$ y esto implica que las conversiones de octal y hexadecimal a binario pueden hacerse de forma muy sencilla agrupando dígitos de a 3 y 4 unidades respectivamente. Los números que se piensan en binario suelen escribirse en hexadecimal para reducir el número de dígitos.

3.1. Datos en la memoria

Si por ejemplo en la memoria RAM tuviéramos la sucesión de bytes `[01010010, 10100001, 01010100, 10100001]` esto podría representar diferentes hipotéticos valores según cómo interpretáramos esos valores.

Por ejemplo, si dijéramos que esos valores representan números enteros positivos representados por su codificación binaria en la memoria estarían (en decimal) los valores `[82, 161, 84, 161]`. Ahora bien, si dijéramos que son números signados, donde el primer bit representa el bit de signo y la codificación es complemento a 2 podría interpretarlos como `[82, -95, 84, -95]`.

La interpretación no termina ahí. Podría decir que cada valor representa un carácter según una tabla, por ejemplo la tabla ASCII Latin 1 y en ese caso tendría `['R', '¡', 'T', '¡']` y esto suena arbitrario, pero es exactamente así cómo se almacenan textos en una computadora. Tanto en la memoria RAM como en la memoria de disco los datos no son otra cosa que valores binarios con una determinada codificación.

Retrocedamos un poco sobre el problema, si la memoria almacena bytes, ¿esto significa que no puedo almacenar números de más de 8 bits?, o sea, tengo sólo $2^8 = 256$ valores posibles para almacenar? No, tranquilamente puedo combinar los 4 bytes del ejemplo y decir que tengo un único número de 32 bits que representa el valor `[1386304673]`.

¿Cuál es la conclusión de esto? En primer lugar la memoria es sólo un dispositivo para almacenar datos en crudo, a la memoria no le interesa qué representa lo que almacenó ni cómo se opera. En siguiente lugar, si sólo pudiéramos ver los datos en una memoria no nos dirían mucho porque los mismos datos pueden tener diferentes interpretaciones según el contexto. El que le va a dar coherencia a esos datos es el compilador. El compilador es el que sabe cuál es el tipo del dato que guardó en determinada posición, y por lo tanto sabe qué instrucción tiene que utilizar el procesador para operar sobre ese dato.

Para nosotros el uso de la memoria va a ser transparente. Si nosotros guardamos el valor entero `1234` recuperaremos el valor entero `1234`. Si a ese número le sumamos uno, vamos a obtener el `1235`. Y no importa si ocupó un byte, dos, cuál fue la codificación para ordenar esos bytes y si se guardó en binario o en otra base. Ese es un problema del procesador y del compilador, para nosotros son números decimales que podemos manipular con las reglas que conocemos.

3.2. Declaración de una variable

Como ya mencionamos, los datos se encuentran en la memoria y son bytes.

La acción de declarar una variable en C es pedirle al compilador que reserve una porción de una determinada cantidad de bytes de memoria y le asigne a esa porción un nombre con el cual nos vamos a referir.

Cuando escribimos `int a;` estamos diciéndole al compilador que identifique con la etiqueta `a` un bloque de memoria de tamaño suficiente como para operar con una variable de tipo entero.

Si cuando declaramos una variable no la definimos eso significa que la variable `a` contendrá el valor que tenía la memoria antes de decir "esta porción de memoria se llama `a`". Consideramos a ese valor como "basura" dado que puede valer cualquier cosa. Esto no es un problema siempre y cuando no pretendamos leer el valor de la variable `a` antes de definirlo.

3.3. Tipos de C

El lenguaje de programación C trae una colección de tipos básicos. Estos tipos se separan en dos categorías: enteros y de punto flotante.

Los tipos enteros son: **`char`**, **`short`**, **`int`**, **`long`**, mientras que los tipos de punto flotante son **`float`** y **`double`**.

3.3.1. Enteros

Los tipos enteros contienen una cantidad fija de bits y en esa cantidad de bits se representan números binarios enteros. Como la cantidad de bits es fija los números tienen una determinada capacidad, un valor máximo, por encima de ese valor ya no pueden seguir guardando información. Lo importante es que por debajo de ese valor pueden guardar cualquier valor particular.

Si tenemos n bits para guardar información y cada bit puede guardar dos valores diferentes, la cantidad de valores difentes que podemos almacenar en los bits será 2^n.

Ya vimos que existen números sin signo y números con signo. En el caso de números con signo, esos 2^n valores se repartirán entre 0 y $2^n - 1$. En el caso de los números sin signo esos valores estarán entre -2^{n-1} y $2^{n-1} - 1$. La asimetría en los signados es porque existe un único cero y es positivo.

Los tamaños relativos de las diferentes clases de enteros dependen de la plataforma y del compilador y no están definidos en el estándar.

Daremos como ejemplo las clases en el compilador GCC para 64 bits[1]:

Tipo	Bits	Bytes	Desde	Hasta
`signed char`	8	1	-128	127
`unsigned char`	8	1	0	255
`short`	16	2	-32.768	32.767
`unsigned short`	16	2	0	65.535
`int`	32	4	-2.147.483.648	2.147.483.647
`unsigned int`	32	4	0	4.294.967.295
`long`	64	8	-9.223.372.036.854.775.808	9.223.372.036.854.775.807
`unsigned long`	64	8	0	18.446.744.073.709.551.615

Lo importante de estos valores es notar que como el crecimiento es exponencial duplicar la memoria incrementa extraordinariamente el rango de los valores.

3.3.2. Tamaño de la memoria

Como ya se dijo, el tamaño de los tipos dependerá de la plataforma y del compilador.

En las aplicaciones donde necesitemos conocer el tamaño de un tipo o de una variable podemos utilizar el operador `sizeof`.

Aplicar `sizeof(x)` siendo `x` un tipo o una variable evaluará al tamaño **en bytes** de `x`.

Para los tipos enteros el único tamaño definido en el estándar es el de un `char`:

`sizeof(char)` $= 1$.

Para el resto de los enteros se verifica que:

`sizeof(char)` $\leq$ `sizeof(short)` $\leq$ `sizeof(int)` $\leq$ `sizeof(long)`, sin más restricciones adicionales.

3.3.3. Desbordamiento (*overflow*)

Como se dijo los enteros tienen un número fijo de bits. ¿Qué pasa cuando una operación necesita más? Sencillamente el resultado se almacena en los bits que haya disponibles.

Por ejemplo, si tuviéramos una variable `unsigned char x = 255;` la misma se representa en binario como `11111111`, es el número más grande que podemos almacenar ($2^8 - 1$). Si hiciéramos `x++` en el mundo de las matemáticas el resultado debería ser `100000000`, es decir 256. Pero como sólo disponemos de 8 bits se almacenará `00000000`, es decir, el resultado será cero. Podemos pensar como que la variable "pegó la vuelta".

Esto aplica también para números signados incluso sin llegar a quedarnos cortos de bits. Si tuviéramos `signed char x = 127;` esta variable se representa como `01111111` ($2^{n-1} - 1$). Si hiciéramos `x++` obtendríamos `10000000`, lo cual entra perfectamente en 8 bits... pero estamos invadiendo el bit que reservamos para el signo, un número que tenga en 1 el bit más pesado tiene que ser un número negativo, por lo que el resultado da -128, es decir el número más chico. También la variable "pega la vuelta".

El desbordamiento puede darse haciendo operaciones o también asignando resultados en una variable de tamaño insuficiente. En todos los casos se guardarán sólo los bits para

[1] Y además utilizaremos esos valores como referencia en el resto del curso, pese a que sepamos que pueden variar en otra plataforma.

los que haya capacidad y se perderán el resto. A diferencia de lo que se puede pensar, este comportamiento es totalmente determinístico y predecible. Tal vez desde la semántica no tenga sentido, pero sintácticamente el lenguaje funciona así.

3.3.4. Números de punto flotante

Los números de punto flotante en C utilizan una representación científica de mantisa fija y exponente.

Por ejemplo, el número $-12{,}345$ podría expresarse con una mantisa fija de 6 dígitos y en base decimal como 123450×10^{-4}. Notar que la notación científica permite escribir números muy chicos o muy grandes jugando con el exponente, pero manteniendo siempre fija la precisión. Todos los números tienen la misma cantidad de cifras representativas. Los números científicos dan cuenta de la magnitud, no del valor exacto.

Los números de punto flotante utilizan internamente algo similar a esto, reservan un bit para el signo, una cantidad de bits para la mantisa y otra para el exponente... con la particularidad de que están en base dos, normalizados y un montón de detalles de implementación que vienen del estándar IEEE 754 y no nos importan.

Nos importa sólo este resumen:

Tipo	Bits	Bytes	Desde	Hasta	Precisión (decimal)
`float`	32	4	$\pm 1{,}4 \times 10^{-45}$	$\pm 3{,}4 \times 10^{38}$	7
`double`	64	8	$\pm 2{,}2 \times 10^{-308}$	$\pm 1{,}7 \times 10^{308}$	16

Y más que todo nos importa de esta tabla la precisión, que está expresada en dígitos decimales: 7 dígitos para el `float`, 16 dígitos para el `double`.

Para dar un ejemplo de números de punto flotante:

```
1 float pi = 3.141592;
2 double pi = 3.141592653589793;
```

Definir cualquiera de las variables con más dígitos no va a aportar información adicional porque se escapan de la representación de la mantisa.

Dado que los números flotantes pueden ser absurdamente chicos o grandes, los literales pueden expresarse en notación científica: `12.1E5` representa $12{,}1 \times 10^5$, es decir `1210000`.

3.3.5. `void`

En la declaración de una función ya vimos que la firma tiene que indicar el tipo de retorno y de los parámetros. ¿Cómo hacemos en el caso de las funciones que no reciben parámetros o las que no devuelven valores (también llamadas "procedimientos")?

```
1 int f(void) {
2     // No recibe parámetros pero devuelve un entero
3     return 8;
4 }
5
6 void g(void) {
7     // No recibe ni devuelve nada
8 }
```

`void` no es un tipo, es la palabra que utilizamos para indicar que no hay devolución o recepción de ningún valor.

3.4. Literales

En C todo tiene tipo, incluso los literales. Generalmente no le prestaremos más atención al tema, pero vale la pena mencionarlo:

```
97      // int
97U     // unsigned int
97L     // long
97LU    // unsigned long
97.0    // double
97.0F   // float
'a'     // int (ASCII 97 = 'a')
'\x61'  // int (0x61 hex = 97 decimal)
```

3.5. Operación entre tipos

En un procesador no se admiten operaciones entre tipos mixtos. Todas las operaciones se hacen siempre entre dos valores del mismo tipo. Asímismo el resultado de operar entre dos valores del mismo tipo es del tipo de los operandos.

Es decir `1 + 2` evaluará a `3` y `1.0 + 2.0` evaluará a `3.0` e internamente el código máquina utilizará dos operaciones totalmente diferentes del procesador, en un caso la operación de suma de enteros y en el otro caso la de flotantes de doble precisión.

Tal vez pasó desapercibido el ejemplo pero en la tabla de operadores se dijo que `7 / 2 = 3` y esto cumple con la norma de que el resultado pertenece al tipo de los operandos. Es más:

```
double a = 7 / 2;
// a = 3.0
```

¿Por qué?, por asociatividad de las operaciones, la expresión `7 / 2` se evalúa primero y evalúa siempre a `3` independientemente del contexto.

Cuando hay dos operandos de diferente tipo entonces el compilador *promueve* el operando de tipo "más chico" al tipo del operando de tipo "más grande". Por ejemplo, en la expresión `1 + 1.2` se tiene un literal de tipo **`int`** y un literal de tipo **`double`**. El compilador considera que el tipo `doube` le gana al tipo **`int`** por lo que promueve el `1` a `1.0`. Luego evalúa la expresión `1.0 + 1.2` la cual evalúa en `2.2`.

El orden de las promociones es el siguiente:

`char`, **`short`** → **`int`** → **`unsigned int`** → **`long`** → **`unsigned long`** → **`float`** → **`double`**

lo cual sigue el orden esperable, tal vez con la salvedad de que los **`unsigned`** le ganen a los **`signed`**.

En general podemos pensar que al promover dado que se pasa a un tipo "más grande" no hay pérdida de información, aunque esto no es del todo cierto. Por ejemplo: `12345678 + 1.0f` → `1234567e1f + 1.0f` → `1234567e1f`[2]. No nos olvidemos de que los enteros guardan números más chicos pero con precisión completa a diferencia de los flotantes que guardan números grandes pero con poca precisión.

Retomando: Por lo general podemos pensar que al promover no hay pérdida de información.

Por el otro lado cuando se convierte un valor de un tipo "más grande" a uno "más chico" se habla de truncamiento.

[2]Tomar este ejemplo como un ejemplo conceptual, dado que los **`float`** operan en base 2 la cantidad de cifras decimales no son *exactamente* 7 si no que depende de cómo es la conversión a binario del valor puntual que además hará aparecer dígitos parásitos. Quedarse con la idea, el resultado real de la operación es mucho más complejo.

El truncamiento puede o no tener pérdida de información, de hecho ha habido un montón de ejemplos hasta el momento en este apunte de truncamiento, como por ejemplo `signed char x = 127;` donde pusimos un literal de tipo `int`, es decir de 32 bits, en una variable de 8 bits. Aquí no hay pérdida de información porque la representación binaria de 127 en 32 bits es `00000000 00000000 00000000 01111111`, es decir, los bytes que se descartan valían todos cero. Análogamente cuando dijimos `float pi = 3.141592;` estamos guardando un literal `double` dentro de un `float`, pero alcanza tanto el rango como la precisión.

En C el truncamiento únicamente va a darse al realizar asignaciones sobre variables más pequeñas. La operatoria entre expresiones sólo va a generar promociones.

Un ejemplo de promociones y truncamientos:

```
1 int sumar(int a, int b) {
2     return a + b;
3 }
4 // ...
5 double x = sumar(4.8, 5.7);
```

El parámetro `a` (`int`) de la función se inicializa con la expresión literal `4.8` (`double`), es decir, hay un truncamiento a `4`. Análogamente `b = 5`. Luego la expresión `a + b` evalúa a `9` y como la función es de tipo `int` se devuelve ese valor. Como `x = 9` ahí hay una promoción, dado que queremos guardar un `int` en un `double`. Finalmente `x` vale `9.0`.

Paréntesis al margen, cuando se habló de declaración de funciones se utilizó de ejemplo esta misma función `sumar()`. La necesidad de que el compilador conozca la firma de las funciones al momento de compilar una llamada a función es poder ajustar todas las conversiones de tipos a los que la función requiere. Si al invocar a `sumar(4.8, 5.7);` no se hubiera provisto la declaración de la función, el compilador adivinaría que la firma es `int sumar(double, double);` y haría fallar la compilación al no encontrar una definición que respete esa firma.

3.6. Conversión explícita de tipos

En C podemos forzar la conversión de tipos si lo necesitamos. Esta operación se conoce como *cast* (del inglés "amoldar") o (castellanizando) *casteo*. Para castear[3] una expresión se antepone el tipo deseado entre paréntesis. Eso va a forzar a la expresión a convertirse al tipo deseado:

```
1 int x = 5;
2 int y = 2;
3
4 int a = x / y;                // a = 2
5 float b = x / y;              // b = 2.0f
6 float c = (float)x / y;       // c = 2.5f
7 float d = x / (float)y;       // d = 2.5f
```

3.7. Redefinición de tipos

Supongamos que nos piden escribir el programa para un censo y tenemos que definir qué tipo de variables vamos a utilizar para almacenar los números. Por ejemplo, si el censo fuera

[3] Sí, a los argentinos nos gusta particularmente no sólo incorporar palabras en inglés si no convertirlas en verbos y después conjugarlas: yo googleo, tú copypasteas, él postea, nosotros ghosteamos, etc. El resto de los países de habla hispana nos suele mirar raro.

un censo barrial me alcanzaría con variables de tipo **short**, si fuera un censo nacional de tipo **int** pero si fuera mundial necesitaría **long**. Analizando el problema me surje que para las necesidades actuales un tipo es más que suficiente pero esas especificaciones podrían cambiar a futuro.

Lo que podemos hacer es crear un nuevo tipo que nos permita abstraernos del tipo:

```
1 typedef unsigned short cantidad_t;
```

Esta sentencia crea un "nuevo" tipo `cantidad_t` que circunstancialmente está declarado como **unsigned short**. Luego podemos declarar variables de este tipo:

```
1 cantidad_t habitantes_san_telmo = 25969;
```

Si eventualmente necesitáramos redefinir el tipo porque el dominio de mi problema se modificó, sólo cambiaríamos la especificación en la línea del **typedef** por lo nuevo. Automáticamente todas las variables de tipo `cantidad_t` se actualizarán.

Ahora bien si bien a veces la necesidad de la redefinición de tipos es para anticiparse a futuros cambios de especificación, muchas este tipo de construcciones simplemente sirven para obtener más abstracción en mi código.

Por ejemplo podríamos asumir que ninguna persona en el corto plazo va a vivir más de 127 años y más aún más de 255, por lo tanto si tuviéramos que declarar variables para almacenar edades tranquilamente podríamos utilizar alguna variante de **char**. Ahora bien, en mi programa no todas las variables de tipo **char** van a representar una edad y las edades no van a destacarse como un tipo en sí. Si quisiéramos ganar en abstracción

```
1 typedef unsigned char edad_t;
```

generaría un nuevo tipo para abstraer el tipo base de las edades. No hay necesidad de redefinir el tipo a futuro, simplemente consideramos que es más abstracto declarar una variable `edad_t edad_juan;` que **unsigned char** `edad_juan;`.

3.8. printf()

Hemos dicho que el primer parámetro de `printf()` es una cadena de formato la cual la función utiliza para saber qué cosas va a imprimir después. También hemos visto que utilizando `"%d"` como formato podíamos imprimir números. Ahora bien, no dijimos que `"%d"` es exclusivamente para imprimir números **int** en formato decimal.

`printf()` tiene una amplia variedad de modificadores de formato, como por ejemplo:

```
1 printf("%d␣%f␣%c␣%s\n", 42, 3.14, 'x', "hola");
```

define el formato de un **int**, un **float/double**, un carácter (los cuales son **int** lo vimos cuando vimos literales) y una cadena de caracteres. La salida de esta línea será `"42␣3.140000␣x␣hola\n"`.

Estos no son los únicos, no sólo hay más modificadores si no que hay un montón de opciones que se pueden manipular en la cadena de formato para imprimir los números alineados, ocupando cierta cantidad de dígitos, rellenando con ceros, con el signo explícito, etcétera, etcétera, etcétera. En el sitio web del curso hay un apunte llamado "Los Secretos de `printf()`" de Don Colton, el cual cubre exhaustivamente las diferentes opciones disponibles. Ese apunte es de lectura obligatoria.

Capítulo 4

El proceso de compilación

Si bien hasta ahora hablamos de la compilación como si se tratara de un proceso monolítico, esto no es así en C. El proceso de compilación consta de varias etapas diferentes y necesitamos entender cada una de ellas para construir nuestro programa y solucionar los problemas que surjan.

El proceso de compilación se divide en tres etapas diferentes: Preprocesamiento, compilación y enlace. Entre las 3 se produce la transformación de nuestro fuente `.c` a un ejecutable.

Preprocesador: El preprocesador es el responsable de la etapa previa a la compilación. Es un programa sencillo que principalmente sabe hacer reemplazos y activar u ocultar fragmentos del código. Las instrucciones del preprocesador empiezan todas con **`#`**. Por ejemplo ya vimos la instrucción **`#include`**, la misma busca el archivo con la ruta indicada, lo abre, copia su contenido y lo pega completo en el lugar en el que estaba la línea del **`#include`**. Ya vimos que eso servía para, entre otras cosas, traer la declaración de la función `printf()`. Al proceso de preprocesamiento entra un fuente `.c` limpio, como lo generó el programador, y sale un fuente expandido con cosas autogeneradas e información de las bibliotecas.

Compilador: El compilador es el responsable de traducir código fuente en código máquina. El compilador analiza sintácticamente cada una de las instrucciones sentencias de nuestro programa y decide la mejor secuencia de pasos de assembly que resuelven eso en nuestra arquitectura. La salida del compilador se llama "código objeto" y es ya prácticamente código máquina. Es importante destacar que durante la compilación sólo el código que nosotros escribimos es compilado.

Enlazador: El enlazador o *linker* es es programa encargado de generar el ejecutable final. El código objeto que compilamos tiene nuestra parte del programa, pero para que nuestro programa sea funcional seguramente utilizamos funciones de biblioteca que nosotros no implementamos. El enlazador puede tomar múltiples códigos objeto y bibliotecas y estructurar un único programa. En ese proceso *enalaza* las llamadas a función que nosotros hayamos hecho, por ejemplo `printf()` con el lugar donde esté realmente el código máquina de dicha función. Además en este proceso es que el programa se estructura para ser un ejecutable, es el enlazador el que define el punto de entrada y verifica que haya un y sólo un `main()`.

La idea de que el proceso de compilación de C no sea monolítico permite estructurar proyectos de software más complejos que veremos mucho más avanzados en la materia. A esta altura es importante sí entender cómo es que nuestro código se integra con las utilidades del compilador. En la etapa de preproceso incluímos los archivos de encabezados `.h` que nos dicen cómo es la firma de las funciones de biblioteca, lo cual ya es suficiente para encarar la

compilación, mientras que el código máquina de esas funciones se incorpora recién en el último paso de enlace donde recibimos código en forma de código objeto `.o` o de bibliotecas `.lib`, `.so`, `.dll`, etc.

Cabe destacar que el único proceso realmente caro de la compilación es justamente la compilación. Tanto el preprocesador son programas muy sencillos que hacen operaciones rutinarias. Este diseño de C que permite compilar pequeños fragmentos de forma individual y luego juntarlos hace que el proceso sea muy eficiente.

4.1. Interpretando la salida del compilador

Parte importante de la programación en cualquier lenguaje es entender al compilador respectivo. En cierta medida es el compilador el que tiene la última palabra al respecto de la sintaxis de un código y el compilador intenta decirnos qué es lo que no es correcto en nuestro programa.

Miremos el siguiente código:

hola.c

```c
#include <stdio.h>

int main(void) {
    print("Hola mundo\n");
    return 0;
}
```

El mismo tiene un error de tipeo, escribimos `print()` en vez de `printf()`. Ahora tomate el tiempo que necesites, poniendo sobre la mesa todas las cosas que ya presentamos en este apunte, para pensar qué y cómo va a fallar durante el proceso de compilación. ¿Qué va a hacer el preprocesador?, ¿qué va a hacer el compilador?, ¿qué va a hacer el enlazador?, ¿vamos a llegar a la etapa de compilación o de enlace o muere primero?

Pensalo, en serio. Es la forma de autoevaluar si estás entendiendo el contenido, o sea, si podés bajar la teoría a la práctica. Si no tenés ni idea de lo que estamos hablando leer la explicación que viene a continuación no te va a aportar nada desde el punto de vista pedagógico. Esto aplica para este ejemplo y para todo el curso.

Bueno, retomando, nada que pongamos en nuestro código debería afectar al procesador, a menos que queramos incluir un archivo inexistente o algo por eel estilo.

El compilador va a atacar nuestro código, ¿hay algo malo con la sintaxis de nuestro código? La realidad es que no, tenemos un `main()` bien estructurado que llama a una función `print()`. Sí hay un detalle importante, nadie proveyó la definición de la función `print()` porque no forma parte de los prototipos que importamos desde `stdio.h` ni tampoco dimos una definición o declaración de función. ¿Qué hacía el compilador cuando se topaba con funciones que no conocía? Eso ya lo vimos: asumía de lo que veía, en este caso va a asumir que la función tiene firma `int print(char *);`[1], es decir, una función que recibe una cadena de caracteres y devuelve un entero. Una vez hecha esa asunción el compilador generará el código para llamar a esa función que se imaginó.

Eso sí, si bien el compilador puede hacer su trabajo, probablemente considere que tiene que avisarme de la decisión que tomó. En este caso, el GCC dirá algo del estilo de:

```
hola.c: In function ‘main’:
```

[1] ¿Qué es esa cosa de `char *`? Cuando estamos aprendiendo la curiosidad es súper positiva, pero en esta materia vamos a decepcionarte mucho al punto que no vas a querer sentirla. La respuesta a cualquier cosa nueva la mayor parte de las veces va a ser "Te juro que no querés saber". Si te hace feliz, digamos que `char *` *vendría a ser* el tipo de las cadenas de caracteres.

```
hola.c:4:5: warning: implicit declaration of function ‘print’; did you mean
                                        ‘printf’? [-Wimplicit-function-declaration]
     print("Hola mundo\n");
     ^~~~~
     printf
```

Apunte al margen: El idioma de la ingeniería es el inglés. Esto no es una preferencia nuestra, lamentablemente es lo que hay. Si todavía no sabés inglés, deberías empezar a aprenderlo. No lo vas a necesitar para esta materia, pero lo vas a necesitar en todas las materias más avanzadas de la carrera. En nuestro caso no es que el compilador tiene infinitos mensajes en la mochila, te vas a topar con una veintena y más difícil que el inglés es aprender qué cosas rotas en nuestro código son las que disparan cada uno de esos mensajes y cómo solucionarlos.

Lo primero al respecto de este mensaje es que es una advertencia (*warning*) no un error. El mensaje dice "declaración implícita de la función `print`" lo cual significa "nadie me dió su firma, estoy adivinando". Además nos dice "¿no quisiste decir `printf`?", cuidado con esas sugerencias, la mayor parte de las veces no son correctas.

Retomando, el compilador genera un código objeto con el código máquina necesario para llamar a `int print(char *);` y es el enlazador el que tiene que buscar ese código máquina, el cual obviamente no existe.

Esta es la salida a continuación de la compilación:

```
/tmp/ccaNYwA8.o: En la función ‘main’:
hola.c:(.text+0x11): referencia a ‘print’ sin definir
collect2: error: ld returned 1 exit status
```

`ld` es particularmente el ejecutable del *linker* del GCC. Los mensajes de enlazador se distinguen rápido de los del compilador en que los de compilador conocen el código fuente, es decir, nos dicen "en la línea tanto de tal código encontré esta sintaxis", ahora bien, al enlazador le llega código máquina, no conoce nuestro código fuente, por lo que sus mensajes son más del estilo de "en este archivo, en tal posición de memoria", en este caso en `hola.c:(.text+0x11)`. El mensaje es claro: No existe `print()`. Más allá de que acá sabemos cuál es el error desde el principio no encontrar una función puede ser un error de tipeo, puede ser que la función no existe, puede ser que me olvidé de incluir alguna biblioteca (biblioteca $\neq$ encabezado).

Observación al margen: Nos está diciendo literalmente `ld devolvió1 como estado de salida`. Ahí tienen la convención `return 0;` $\Longrightarrow$ todo bien, cualquier cosa diferente de `0` $\Longrightarrow$ error.

4.2. Parámetros del compilador

En nuestro primer acercamiento al compilador compilamos el hola mundo sencillamente como

```
$ gcc hola.c -o hola.exe
```

ahora bien, hay más cosas que prestar atención en el proceso de compilación y este es el momento de introducirlas.

4.2.1. Estándar

Como ya dijimos en este curso vamos a desarrollar C según el estándar ISO-C99. ¿Por qué es que es importante programar dentro de determinado estándar? Se trata de un tema de portabilidad. Nosotros queremos escribir en un lenguaje de alto nivel porque queremos

independizarnos de la plataforma. Para poder independizarnos de ella necesitamos tener compiladores para las plataformas donde queramos ejecutar nuestro código. Utilizar estándares garantiza eso, no vamos a atarnos a lo que, por ejemplo, el compilador GCC considere que es el lenguaje C sino que vamos a decirle al GCC que nuestro código es compatible con el estándard C99. Si quisiéramos cambiar de compilador a cualquier otro sólo deberíamos verificar que sea uno que cumpla con ese estándar y ya.

El GCC particularmente conoce muchos estándares y no sabe a priori cuál queremos utilizar nosotros. Cosas que en una versión de C son correctas en otra no, incluso hay comportamientos que son diferentes según la versión. Queremos asegurarnos de no introducir cosas que no pertenezcan a C99.

Para eso debemos agregar en la línea de compilación `-std=c99`, y más aún, obligamos a un cumplimiento más estricto agregando `-std=c99 -pedantic`.

4.2.2. Advertencias

Las advertencias del compilador nos avisan de posibles errores semánticos, dado que si fueran sintácticos no habría compilación. El 99,9 % de las veces que el compilador identifica algo para advertirnos eso se corresponde con un error en nuestra lógica. Las advertencias son importantes y en este curso no aceptaremos código que tenga advertencias al compilar.

Ahora bien, el compilador elige sobre qué cosas mostrarnos advertencias o no. Para activar todas las advertencias del compilador tenemos que agregar `-Wall`.

Si quisiéramos ser mucho más estrictos podríamos pedirle al compilador que directamente trate a las advertencias como si fueran errores con `-Werror`.

4.2.3. Biblioteca matemática

Por razones históricas la biblioteca de C está partida en dos: La `libc` y la `libm`. La `libm` tiene todas las funciones de manipulación de números de punto flotante, es decir, la biblioteca matemática, mientras que la `libc` tiene el resto. En muchas versiones de compilador por omisión no se enlaza con la `libm`.

Para agregar a la biblioteca matemática al momento de enlace hay que agregar `-lm` como último parámetro de la línea de compilación.

4.2.4. Entonces

La línea completa recomendada para compilar un programa es:

```
$ gcc hola.c -o hola -std=c99 -pedantic -Wall -lm
```

4.3. Constantes

En nuestro código muchas veces necesitamos tener valores de constantes. Las contasntes a veces pueden ser valores universales como por ejemplo el valor de π y otras veces pueden ser cosas constantes a nuestro programa, como por ejemplo la cantidad de países en el mundo, que si las quisiera modificar debería recompilar mi programa.

Tomemos de ejemplo π, sabemos que un **`float`** soporta aproximadamente 7 dígitos decimales representativos. Es decir cada vez que necesitemos el valor de π deberíamos escribir `3.1415926f`[2]. Escribir más números sería innecesario, escribir menos introduciría error en las

[2]Pusimos 8 dígitos porque ya se dijo es aproximadamente 7, no sabemos si el último se contabiliza, pero es preferible estar del lado de la seguridad.

operaciones. ¿Vamos a escribir ese número cada vez?, ¿y si mañana decidimos migrar de **float** a **double** tenemos que reemplazar todas las ocurrencias por ese valor?

Reemplazar las ocurrencias de `3.1415926` por otro valor es más o menos automatizable. Ahora bien, ¿qué pasaría si hicimos un programa en un momento en el que el mundo tenía 100 países y, como la provincia de Córdoba finalmente se independizó, deberíamos actualizar ese número a 101? ¿Estamos seguro que todo `100` en mi programa se correspondía con la cantidad de países y no es resultado de otra cosa, como por ejemplo, la nota máxima para calificar en Química, el punto de hervor del agua en grados Celsius, la conversión entre pesos y centavos o el cálculo de un porcentaje?

Bueno, para todas esas cosas necesitamos constantes. La idea es definir una constante una única vez, en un solo lugar, ponerle un nombre y usar ese nombre donde haga falta. No sólo va a mejorar la mantenibilidad si el día de mañana quiero modificar una cantidad, además va a ganar mucho en legibilidad porque en una fórmula veré un nombre que me habla del número de países en vez de una cifra suelta que no sé qué significa.

El lenguaje C provee dos maneras diferentes de generar constantes. Una es dentro del código de C y la otra es mediante el procesador.

4.3.1. Variables constantes

```
1 const float pi = 3.1415926;
```

Declara y define una variable de tipo **float** que se llama `pi` y es constante. ¿Variable constante no es un oxímoron?...

Repasemos el concepto de declarar y definir una variable: El compilador busca memoria del **sizeof** necesario. Le pone un nombre a esa memoria. Escribe en esa memoria la representación binaria de ese valor. Luego cada vez que en una expresión utilice `pi` eso evaluará a ir a buscar ese valor a la meoria.

Una variable constante tiene todo eso, con la restricción de que tengo que definirla en tiempo de declaración y que luego no puedo modificar ese valor. Cualquier intento que haga de redefinir `pi` será un error de compilación.

En C se pueden declarar variables constantes y esto puede hacerse fuera de las funciones, al comienzo del programa y esas variables estarán disponibles para utilizar en todas las funciones.

Ahora bien, esta no es la manera más común de resolver el tema de los valores constantes en C.

Aclaración: En este curso no se permite bajo ningún punto de vista la existencia de variables fuera de las funciones (i.e. globales) que no sean de tipo **const**.

4.3.2. Etiquetas

Dijimos ya varias veces que el preprocesador es una máquina de reemplazar cosas por cosas. La instrucción

```
1 #define PI 3.1415926f
```

define a la etiqueta `PI` con el valor que sigue a continuación (notar la ausencia de operador de asignación y de punto y coma). En el proceso de preproceso cada vez que el preprocesador vea la etiqueta `PI` en el código la reemplazará por el valor literal `3.1415926f`[3].

Al compilador no le llegará el `PI`, le llegará directamente `3.1415926f`. No hay memoria asociada, no hay variables, no hay nada, sólo lo mismo que si hubiéramos escrito el valor de π donde lo necesitábamos.

[3] ¿Por qué pusimos la `f` en la etiqueta pero no en la variable constante? Porque en el caso de la variable al ser la variable **float** la asignación se trunca a ese tipo y al utilizar `pi` se utilizará como **float**. En el caso de la etiqueta al ser un literal suelto si queremos que sea **float** tenemos que darle el tipo al literal.

Convención **(importantísimo)**: En C se utilizan mayúsculas si y sólo si estamos declarando una etiqueta. Esta es una convención fuertísima del lenguaje. Si vemos un identificador en mayúsculas vamos a asumir inmediatamente que se trata de un `#define`. **Todas** las etiquetas tienen que estar en mayúsculas y **todos** los identificadores de variables y funciones en minúsculas.

Y con las etiquetas completamos la colección: Los archivos de encabezados de C contienen básicamente declaraciones de funciones, redefiniciones de tipos con `typedef` y finalmente etiquetas con `#define`. Eso es lo que traemos de la biblioteca cuando hacemos un `#include`. Notar que son todas cosas que no definen funcionalidad si no que le avisan al compilador de qué cosas tiene disponibles para utilizar.

Capítulo 5

Control de flujo

Hasta ahora desarrollamos programas donde de forma secuencial se ejecutan todas y cada una de las líneas del mismo una única vez. Llamamos control de flujo a las estructuras del lenguaje que nos permiten repetir instrucciones o ejecutar bloques de código de forma condicional.

5.1. El ciclo `while`

El ciclo **`while`** nos permite repetir un bloque de código mientras una condición sea verdadera. Por ejemplo:

```
1  #include <stdio.h>
2
3  int main(void) {
4      int i = 1;
5      while(i <= 10) {
6          printf("Hola\n");
7          i++;
8      }
9      printf("Chau\n");
10     return 0;
11 }
```

El encabezado de la instrucción **`while`** lleva entre paréntesis una condición. Cada vez que se ejecute el **`while`** se evaluará esa condición, si la misma fuera verdadera entonces se ejecutará el bloque siguiente. Luego de ejecutar el bloque se evaluará la condición y se seguirá repitiendo esta secuencia. Si al evaluar la condición la misma fuera falsa, se seguirá ejecutando lo que siga a continuación del bloque.

En nuestro ejemplo comenzamos con la variable `i` valiendo `1`. En la primera iteración se preguntará si `i` es menor o igual a `10`. Dado que `1` es menor o igual a `10` la condición será verdadera. Entonces se ejecutará el bloque. El bloque hace dos cosas: Primero imprime `"Hola\n"` y luego incrementa el valor de `i`, por lo que la primera iteración terminará con `i` valiendo `2`. Acto seguido se evaluará de vuelta la condición, como `i` sigue siendo menor o igual que `10` se ejecutará el mismo bloque de nuevo, y esto seguirá pasando 9 veces más hasta que después de imprimir `"Hola\n"` por décima vez `i` será incrementado una vez más y valdrá `11`. Al volver a evaluar `i <= 10` esta vez esa expresión será falsa y el **`while`** terminará. Al terminar el **`while`** se imprimirá `"Chau\n"` y luego terminará el programa.

Más adelante hablaremos de qué características tienen estas expresiones que dan "verdadero" o "falso" y cuáles son los operadores que tenemos disponibles además de <=.

5.2. Bloques

Como se dijo anteriormente después de una instrucción de control de flujo viene un bloque.

Ya vimos cuando hablamos de instrucciones que los bloques en C se delimitan entre `{` y `}`. Ahora bien, esto es para generar bloques de múltiples instrucciones. Una única instrucción también constituye un bloque, por lo que si tuviéramos por ejemplo un **`while`** que podría resolverse con una única línea podríamos omitir las llaves. Por ejemplo:

```
1 int i = 0;
2 while(i < 10)
3     printf("%d\n", i++);
```

imprimirá `0`, `1`, `2`, ... `9` y terminará.

Otra cosa importante de los bloques es la indentación del código. Cuando anidamos un bloque dentro de otro bloque debemos incrementar la sangría. En nuestro ejemplo de la sección anterior la función `main()` constituye un bloque por lo que su código está indentado un nivel de sangría con respecto al **`#include`** o a la declaración de la firma del `main()`. Ahora bien, cuando dentro de la función `main()` iniciamos un bloque **`while`** volvemos a incrementar la sangría de todo lo que está dentro de él.

La sangría es invisible al compilador y a la sintaxis del lenguaje y podríamos no utilizarla. Pero es obligatoria para poder entender el alcance de los bloques de un código fuente al leerlo y no aceptaremos códigos que no estén correctamente indentados. La sangría es tan importante que hay lenguajes posteriores a C que no necesitan las llaves para marcar bloques sino que se guían pura y exclusivamente por la indentación. Es decir, convirtieron algo que era una convención de estilo en sintaxis del lenguaje.

5.3. El ciclo `for`

En programación son muy comunes las iteraciones en las cuales antes de empezar el bucle hay que inicializar un valor y después de ejecutar el bloque hay que hacer una actualización para la siguiente iteración. De hecho el ejemplo de saludar 10 veces que ya hicimos tiene esa estructura.

Dado que este patrón es muy frecuente, el lenguaje C provee una instrucción de flujo que está pensada específicamente para estos casos:

```
1 #include <stdio.h>
2
3 int main(void) {
4     for(int i = 1; i <= 10; i++)
5         printf("Hola\n");
6     printf("Chau\n");
7     return 0;
8 }
```

funciona idénticamente al ejemplo que dimos de **`while`**. La instrucción **`for`** tiene 3 parámetros (¡que se separan con ;!): Una inicialización, la condición de corte y un incremento, en este ejemplo **`int`** `i = 1`, `i <= 10` y `i++` respectivamente.

La única diferencia operativa entre este ejemplo de **`for`** y el ejemplo del **`while`** es que en este caso la variable `i` existe únicamente dentro del **`for`**, mientras que en el otro ejemplo estaba

declarada en el `main()` y era visible en toda la función. Si necesitáramos persistir la variable de iteración fuera de un **`for`** deberíamos declararla antes del mismo.

Si el **`while`** y el **`for`** hacen lo mismo o son intercambiables entre sí, ¿vamos a usar los dos?, ¿vamos a preferir uno sobre el otro? La idea es que **siempre** que tengamos una iteración donde se observe el patrón de inicialización previa e incremento posterior vamos a preferir usar **`for`**. Es un tema de estilo: En la lectura de la instrucción entendemos por completo cuál va a ser el comportamiento completo del bloque, incluso aunque este sea muy largo y no veamos su totalidad. Vamos a dejar el **`while`** para iteraciones más libres donde el comportamiento de la iteración va a depender de cosas menos definidas.

5.4. El ciclo `do-while`

Como con dos iteradores no alcanzaba, existe además el ciclo **`do-while`** que es una variante del **`while`**:

```
1 int n = 7;
2
3 do {
4     printf("%d\n", n);
5     n /= 2;
6 } while(n > 0);
```

Imprimirá `7`, `3`, `1`.

¿Cuál es la diferencia con el **`while`**? Que como primero se ejecuta el bloque y luego se verifica la condición el ciclo **`do-while`** garantiza que el bloque se ejecute al menos una vez. En el ciclo **`while`** si la condición es falsa desde el inicio nunca se ejecutará el bloque. Una vez que se ejecutó la primera iteración el comportamiento de **`while`** y **`do-while`** es el mismo dado que ambos son una sucesión de ejecutar bloque, validar corte.

El **`do-while`** se usa bastante poco, dado que el patrón que plantea no es muy común. Suele ser conveniente por ejemplo para operaciones de interacción con algo externo: Obtengo un dato y mientras el dato no valide determinado criterio vuelvo a pedirlo. Como pedirlo y volver a pedirlo seguramente se haga de la misma forma, con **`do-while`** garantizo que al menos se pida una vez y no duplico el código del pedido como tendría que hacerlo con **`while`**.

5.5. Booleanos

Cuando hablamos de la condición de corte del **`while`**, el **`for`** y el **`do-while`** dijimos que era expresiones que evaluaban a valores de verdad o falsedad. Este tipo de operaciones se conocen como operaciones booleanas, y derivan del álgebra de Boole denominada así por el matemático George Boole (1815-1864) que la definió. En este tipo de álgebra tiene la particularidad de que se opera sobre conjuntos muy pequeños de valores, en nuestro caso únicamente dos.

La lógica booleana es inherente a la electrónica digital, área dentro de la cual se encuentran los procesadores. Lo que define este álgebra es un conjunto de operaciones que nos permite operar sobre dispositivos que admiten dos estados: prendido-apagado, magnetización positiva-negativa, tensión de 0V-5V, etc. que son la base de los circuitos digitales. Particularmente en computación pensamos en estados de verdadero-falso.

5.5.1. Booleanos en pre ISO-C99

Si bien la lógica booleana es parte fundamental de la programación C no tenía un tipo para representar a los booleanos en su comienzo, y en realidad durante casi 30 años desde creado. Si

bien el lenguaje siempre tuvo operadores que devolvían valores booleanos y operaban sobre los mismos, originalmente no se pensó en contenerlos dentro de un tipo.

Para el lenguaje C anterior al estándar C99 cualquier expresión entera podía ser vista como un valor booleano. La convención era sencilla: El valor `0` evaluaba a falso mientras que cualquier otro valor evaluaba a verdadero. Es decir, si uno operaba una expresión booleana como por ejemplo `5 < 10` esta expresión hubiera evaluado a cualquier valor arbitrario diferente a cero, por ejemplo `-4654`. Uniendo esto con las secciones anteriores, básicamente cuando uno utiliza una instrucción de tipo **`while`**`(condicion)` se ejecutará el bloque siempre y cuando la expresión `condicion` evalúe a un número diferente a cero.

Por ejemplo el código

```
1 for(int i = 10; i > 0; i--)
2     printf("Hola\n");
```

tendrá el mismo comportamiento que el código

```
1 for(int i = 10; i; i--)
2     printf("Hola\n");
```

En ambos casos se imprimirá 10 veces `"Hola\n"`. Dado que `i` empieza la iteración en `10` y va a ir decrementando de a una unidad será una sucesión `10`, `9`, `8`, ... Con estos valores la expresión `i > 0` evaluará a verdadero hasta que `i` alcance el valor de `0`, donde evaluará a falso. Del mismo modo, la expresión `i` mirada desde su verdad o falsedad booleana va a evaluar verdadera hasta que `i` valga `0` dado que `0` es el único entero que se considera falso. Ambas condiciones son equivalentes en el contexto de esta iteración[1].

5.5.2. Booleanos en ISO-C99

Cuando en el estándar C99 decidieron introducir finalmente variables de tipo booleano lo hicieron de tal manera de no romper los programas de los 30 años anteriores. Por lo que tomaron una serie de soluciones de compromiso que extienden el comportamiento original de C sin dejar de ser intuitivas para los que codifiquen código ya pensando en el estándar nuevo.

Como bien dijimos independientemente de que el lenguaje C tuviera o no variables booleanas los programadores siempre las utilizaron, por lo que es común que en cualquier código viejo hayan definido algún tipo booleano con **`typedef`** o generado símbolos para verdadero y falso con **`#define`** o equivalentes.

Para no chocar con código del usuario los booleanos se implementan con un tipo nativo llamado `_Bool` (que no utilizaremos nunca, pero mencionamos sólo por completitud).

El cambio que implementa el lenguaje es que ahora los operadores booleanos devuelven `0` o `1`. Es decir, en el estándar C99 la operación `5 < 10` que vimos anteriormente va a devolver `1`. Y básicamente este es el **único** cambio que se introdujo. Todo lo que dijimos respecto al pre ISO-C99 sigue valiendo, cualquier entero puede interpretarse como booleano, cualquier entero diferente de cero se interpretará como verdadero mientas que el cero se interpretará como falso. Pero ahora los operadores devuelven sólo `0` y `1`.

Como dijimos anteriormente, se introdujo un nuevo tipo `_Bool`, pero también dijimos que no íbamos a utilizarlo. En su lugar si vamos a utilizar booleanos lo que haremos será incluir el encabezado `stdbool.h`. El contenido de este encabezado podemos resumirlo básicamente como:

```
1 typedef _Bool bool;
2 #define false 0
3 #define true 1
```

[1] Fuera del contexto de esta iteración pueden no serlo, es decir, la expresión booleana `i` siempre es equivalente a la expresión booleana `i != 0`, no a `i > 0` como en el ejemplo.

O sea, nos da un tipo `bool` y nos da las etiquetas `true` y `false`[2]. Nosotros siempre vamos a utilizar el tipo con este **`#include`**.

5.5.3. Operadores

Como ya vimos, hay operadores que dadas expresiones numéricas nos devuelven valores booleanos, la lista completa de ellos es:

`==`: La igualdad. Ejemplo: `1 + 1 == 2`, evaluación `true`.

`!=`: La desigualdad. Ejemplo: `1 + 1 != 2`, evaluación `false`.

`<`: Menor que. Ejemplo: `1 + 1 < 2`, evaluación `false`.

`<=`: Menor o igual que. Ejemplo: `1 + 1 <= 2`, evaluación `true`.

`>`: Mayor que. Ejemplo: `1 + 1 > 2`, evaluación `false`.

`>=`: Mayor o igual que. Ejemplo: `1 + 1 >= 2`, evaluación `true`.

Todos estos operadores toman expresiones enteras o de punto flotante y devuelven valores booleanos.

Luego están los operadores que toman expresiones booleanas[3] e implementan funciones lógicas sobre ellas. Estos operadores son: `&&` (and), `||` (or) y `!` (not).

El operador `&&` (and)

El operador *and* del álgebra de Boole (de símbolo $\wedge$ en lógica proposicional) *pregunta* si sus dos operandos son verdaderos, o sea si `a` **y** `b` son verdaderos.

Su tabla de verdad es:

a	b	a && b
false	false	false
false	true	false
true	false	false
true	true	true

Es decir, sólo devolverá `true` si ambos operandos son `true`.

El operador `||` (or)

El operador *or* del álgebra de Boole (de símbolo $\vee$ en lógica proposicional) *pregunta* si alguno de sus operandos es verdadero, o sea si `a` **o** `b` son verdaderos (de forma inclusiva).

Su tabla de verdad es:

a	b	a && b
false	false	false
false	true	true
true	false	true
true	true	true

Es decir, sólo devolverá `false` si ambos operandos son `false`.

[2] ¿Cómo?, ¿que las etiquetas no tenían que estar siempre en mayúsculas? Bueno, las que ponés vos sí. El tipo que escribe estándares las pone como quiere.

[3] Y no olvidar que en C las expresiones booleanas pueden ser `true`, `false` y el resultado de operar alguno de los operadores anteriores, pero también cualquier expresión entera en general.

El operador `!` (not)

El operador *not* del álgebra de Boole (de símbolo ¬ en lógica proposicional) es un operador monario que invierte el valor de su operando, o sea devuelve **no** `a`.

Su tabla de verdad es:

`a`	`! a`
`false`	`true`
`true`	`false`

5.5.4. Cortocircuito

Supongamos la siguiente expresión booleana: `1 + 1 == 3 && (! 8 % 6 != 0)`. ¿Podés darte cuenta rápidamente a cuánto evalúa esa expresión en su conjunto? Bueno, la respuesta debería ser sí. `1 + 1 == 3` obviamente evalúa a `false`, ¿y `(! 8 % 6 != 0)`?, bueno, básicamente a nadie le importa a cuánto evalúa esa expresión. Si ya dijimos que el único caso en el que el operador and evalúa a `true` es si ambos operandos evalúan a `true` y ya estamos viendo que el primero de los dos operandos evaluó a `false` sabemos que la expresión completa evalúa a `false`[4].

El lenguaje C entiende el funcionamiento del and y el or y no se toma el trabajo de evaluar el segundo operando si con la evaluación del primero ya le alcanza para definir el resultado de la evaluación global. Este comportamiento se conoce como *cortocircuito*.

En la sección que hablamos de precedencia y asociatividad dijimos que en el lenguaje C no estaba garantizado el orden de evaluación de las expresiones. Bueno, no está garantizado el orden de evaluación salvo para and y para or. En el caso de and y or el orden de **evaluación** es siempre de izquierda a derecha.

Por ejemplo, si tuviéramos:

```
1 while(n < 0 || sqrt(n) > 10) {
2     ...
3 }
```

nunca se calcularía una raíz negativa. Si `n` fuera negativo eso ya es suficiente motivo para concluir hay que ejecutar el bloque del **`while`**. Sólo se computarìa la raíz si `n` fuera positivo y todavía hubiera que evaluar el segundo operando para saber el resultado global del `||`.

5.6. El condicional `if`

Muchas veces en nuestro código queremos ejecutar algo o no, una única vez, dependiendo de una condición. Para esto tenemos la instrucción **`if`**:

```
1  int main(void) {
2      float n;
3      // ...
4
5      if(n < 0) {
6          printf("No␣podemos␣calcular␣raíces␣negativas!\n");
7          return 1;
8      }
9
10     printf("La␣raíz␣de␣%f␣es␣%f\n", n, sqrt(n));
11
```

[4] Y si te queda la duda `(! 8 % 6 != 0)` evalúa a `false`. ¿Por qué?, analizalo, tenés todas las herramientas para hacerlo y es un buen ejercicio.

```
12      return 0;
13  }
```

El **if** fuerza una ejecución condicional, sólo se ejecutará el bloque si la condición evalúa a `true`.

Y dicho sea de paso: Bienvenidos a nuestro primer `main()` que devuelve algo diferente de `0`. No sabemos de dónde salió el valor de `n` pero si mi programa sirve para calcular raices y tengo un valor negativo mi programa no puede hacer lo que tiene que hacer y fallará con un código de error.

Volviendo, **if**`(condicion)` significa "si la `condicion` es verdadera hacé esto". Muchas veces si la condición es verdadera queremos hacer determinada cosa, pero si es falsa queremos hacer otra cosa diferente de forma excluyente. Para eso sirve el **else**. Toda instrucción **if** puede tener un **else** de forma opcional:

```
1  if(n >= 0)
2      printf("%f^0.5 = %f\n", sqrt(n));
3  else
4      printf("%f^0.5 = %fi\n", sqrt(-n));
```

El **else** se interpreta como "si no", es decir, "si `n >= 0` ejecutá el primer bloque, si no ejecutá el segundo". Notar que el **else** al ser excluyente con el **if** es en cierta medida equivalente a un **if**`(! (n >= 0))`, o sea, se entra al **else** si no se entró al **if**... equivalente pero no igual, porque la condición se evalúa una única vez al evaluar el **if**. No importaría que dentro del bloque del **if** se modificara el valor de `n`, si entra al **if** no entra al **else** y viceversa: **Son excluyentes**.

Volvamos dos ejemplos más atrás, al ejemplo del **if**. Verificábamos que `n` fuera positiva para decidir si calcular o no una raíz... ¿no deberíamos haber usado **else** para calcularla? No, porque el bloque **if** terminaba con un **return**. Es decir, si entrábamos al bloque del **if** la función se abortaba, por lo tanto **todo** lo que continúe al bloque del **if** se va a ejecutar sí y solo sí no se entró al **if**. El "si no" está implícito al terminar el **if** con **return**. Y dado que el **else** sería redundante **no** lo escribimos (y en la siguiente sección vamos a argumentar fuertemente en contra de escribirlo).

Dado que el **else** requiere un bloque, ese bloque puede ser cualquier cosa y particularmente puede ser otro **if**. Cuando tenemos condiciones mutuamente excluyentes podemos encadenar tantos bloques **if-else** como queramos. Por ejemplo:

```
1  if(nota < 4)
2      printf("Reprobado\n");
3  else if(nota < 7)
4      printf("Bien\n");
5  else if(nota < 10)
6      printf("Muy bien\n");
7  else
8      printf("Sobresaliente\n");
```

El último **else**, como todo **else** es optativo. Lo importante a remarcar de esta construcción es que cada una de las ramas es excluyente con la anterior. Es decir no es que va a entrar al segundo **if** si `nota < 7` evalúa positivo la condición para entrar es `(! nota < 4)&& nota < 7` o, dicho de otra forma, `nota >= 4 && nota < 7`. Sólo se va a preguntar si la `nota` es menor a `7` si antes se descartó que no fuera menor a `4`.

5.7. *Early return*

Los programas de la vida real frecuentemente tienen que hacer muchas validaciones previo a realizar la acción primaria que necesitan hacer.

Imaginemos que tenemos que realizar una transferencia bancaria:

```
resultado_t transferir(...) {
    resultado_t resultado = EXITO;
    if(existe(origen)) {
        if(existe(destino)) {
            if(saldo(origen) >= cantidad) {
                if(mensaje != VACIO) {
                    _transferir(origen, destino, cantidad, mensaje
                        ↪ );
                } else {
                    resultado = ERROR_MSJ_VACIO;
            } else {
                resultado = ERROR_SALDO;
        } else {
            resultado = ERROR_DESTINO;
    } else {
        resultado = ERROR_ORIGEN;
    }
    return resultado;
}
```

Es decir, para realizar una transferencia necesitamos que exista la cuenta de origen, la de destino, que el saldo sea sufiente y que haya un mensaje[5]; cualquier otro caso será un error.

Ahora bien, este patrón de validaciones anida indefinidamente `if` adentro del `if` anterior de forma sucesiva y se hace muy difícil seguir qué `else` se corresponde con cada `if`.

Miremos qué pasa si cambiamos la estrategia:

```
resultado_t transferir(...) {
    if(! existe(origen))
        return ERROR_ORIGEN;

    if(! existe(destino))
        return ERROR_DESTINO;

    if(saldo(origen) < cantidad)
        return ERROR_SALDO;

    if(mensaje == VACIO)
        return ERROR_MSJ_VACIO;

    _transferir(origen, destino, cantidad, mensaje);
    return EXITO;
}
```

¿No mejora muchísimo la legibilidad?, ¿ahora no es inmediato entender qué validación dispara qué código de error?, Si tuviéramos que agregar una validación adicional, ¿no tendríamos

[5] Y si bien hay un `&&` implícito entre todas las condiciones no podemos usarlo porque si no no podríamos retornar el código de error correspondiente.

que tocar mucho menos código? (donde tocar mucho código implica tener más chances de equivocarse)

¿Qué fue lo que cambiamos entre el primer código y el segundo? Bueno, de eso se trata el patrón de *early return*, o "return temprano": Si ya sabemos el resultado de la ejecución de la función entonces **no** seguimos adelante con la función. Interrumpimos el flujo de la misma inmediatamente. No hace falta esperar.

Como dijimos, los programas de la vida real tienden a requerir múltiples validaciones antes de procesar un resultado. La estrategia para tener un early return es validar *de forma pesimista*, en vez de decir "sigo adelante si esto que quiero validar está bien" decimos "interrumpo si esto que quiero validar está mal". Es tan sencillo como eso.

5.8. Funciones y variables booleanas

Muchas veces tendremos funciones que devuelven valores booleanos o variables que los almacenan. En todos los casos le pondremos a esas entidades nombres que representen una pregunta. De este modo será absolutamente claro qué tendrá que devolver/almacenar esa entidad en un caso o el otro.

Para que se entienda, tenemos una función `bool validar_número_primo(int n);`, ¿cuándo devuelve `true` y cuándo devuelve `false`? En cambio pensemos en la función `bool es_primo(` ↪ `int n);`. La nomenclatura de la función se responde o por sí o por no, hay una verdad o falsedad.

Y más allá de eliminar ambigüedades en convenciones de devolución este estilo de nomenclatura permite ganar en legibilidad de código. Por ejemplo:

```
1 if(es_primo(n))
2     ...
```

Literalmente se lee "si es primo ejecutar el bloque".

Ya que estamos, ¿implementamos la función? Un número es primo si sólamente es divisible por 1 y por sí mismo.

¿Podemos validar que un número sea primo de forma directa? La realidad es que no ésta, al igual que muchas otras validaciones, se realiza de forma indirecta: Lo que podemos verificar es que un número **no** sea primo, si le encontramos divisores otros que 1 y sí mismo. Si para un determinado número no le encontramos ningún divisor, entonces podemos decir que es primo.

Pero si hay infinitos divisores, ¿cómo podemos probar un número contra todos? Bueno, esto es más fácil, podemos acotar el problema. Una cota inicial podría ser darse cuenta de que ningún número mayor a un número puede ser divisor. Esto nos elimina una infinidad de divisores a probar.[6]

Implementemos la función:

```
1 bool es_primo(int n) {
2     for(int i = 2; i < n; i++)
3         if(n % i == 0)
4             // i divide a n
5             return false;
6     return true;
7 }
```

Notemos un par de cosas en esta implementación. La primera es el uso de early return. Si encuentro un único divisor **no** sigo adelante con la iteración: Ya está, ya sé que `n` no puede

[6] ¿Es esta la mejor cota?... rápidamente podrías darte cuenta de que dado que el mínimo resultado que da una división es 2 si no encontramos un divisor menor o igual a $\frac{n}{2}$ no vamos a encontrarlo. Ahora bien, ¿es esta la mejor cota? No, no lo es. Te queda de tarea pensar cuál es.

ser primo, no gasto recursos. Pensémoslo con un ejemplo, ¿podés decidir en un vistazo si el número 12312389122343436 es primo? No importa que tenga chiquicientos dígitos, si termina en un número par es divisible por 2, ya está.

La siguiente cosa a destacar es la inexistencia de `else`. No es `if(n % i == 0)return ↪ false; else return true;`. Para llegar a la conclusión de que el número no es primo no me alcanza con que no sea par (el primer valor de mi iteración) **necesito** probar con todos los valores de `i`. El `return true;` del final se ejecuta únicamente si el `for` termina sin haber entrado **nunca** al `if`.

5.9. `break` y `continue`

Volviendo a los ciclos `while`, `for` y `do-while` hay dos instrucciones que los alteran y que tiene sentido explicar ahora que ya conocemos el condicional `if` dado que se usan en combinación con él.

Los tres ciclos que tiene C se interrumpen si la condición de corte evalúa a `false`. Simple. Ahora bien, hay veces donde hay muchas cosas que deberían interrumpir un ciclo, y meter todas las condiciones de corte como parte de la condición implica hacer código complicado, requerir variables auxiliares que sirvan como centinelas para cortar, etc.

Por fuera de la condición de corte principal, cualquier ciclo se puede interrumpir con la instrucción `break`:

```
1 while(n > 0) {
2     if(n % 42 == 0)
3         break;
4
5     printf("%d\n", n);
6     n -= 3;
7 }
```

La condición de corte principal es `n > 0`, ahora bien, hay una segunda condición que puede interrumpir el ciclo. Si `n` fuera divisible por `42` el ciclo se termina, sin siquiera llegar al `printf()`. Notar que **no tiene sentido** poner el `break` fuera de un `if`, si el `break` estuviera dentro del bloque del `while` no habría iteración dado que se interrumpe siempre a la primera ejecución.

Otras veces sucede que si se da determinada condición queremos saltearnos la ejecución del bloque y "continuar" a la iteración siguiente. Por ejemplo:

```
1 for(int i = 1; i <= 10; i++) {
2     if(i % 3 == 0)
3         continue;
4     printf("%d\n", i);
5 }
```

Imprimiría los números del 1 al 10, salteándose los múltiplos de 3. `continue` interrumpe el bloque y salta al ciclo siguiente.

Así como `break` nos evita tener que complejizar la condición de corte, `continue` nos evita tener que meter el resto del bloque dentro de un gran `else`.

Notar que el código equivalente con `while` debería ser:

```
1 int i = 1;
2 while(i <= 10) {
3     if(i % 3 == 0) {
4         i++;
5         continue;
```

```
6       }
7       printf("%d\n", i);
8       i++;
9   }
```

En el **for** como el incremento está fuera del bloque está garantizado que se ejecute independientemente de si el bloque fue ejecutado en su totalidad o no. En cambio si en un **while** tuviéramos un incremento el mismo no tiene ninguna jerarquía adicional con respecto al resto del bloque, es una instrucción más. Este es uno de los tantos argumentos a favor de usar **for siempre** que tengamos una iteración con un comportamiento de inicio, condición, incremento definidos.

¿Qué pasaría si en el ejemplo anterior olvidáramos el `i++` dentro del **if**? El bucle jamás terminaría, porque el valor de `i` quedaría fijo en un valor múltiplo de `3`. Siempre podemos utilizar la tecla Control + C para matar la ejecución de un programa.

5.10. El condicional `switch`

Hay una construcción de control de flujo del lenguaje C que tiene una aplicación acotada y es el condicional **switch**. Un bloque **switch** nos permite evaluar diferentes valores **exactos** de una variable **entera**. Y las dos palabras que están en negrita en la oración anterior son las que hacen que la aplicación sea acotada: Si no tenemos variables enteras y chequeos exactos no sirve.

La sintaxis es así:

```
1   switch(nota) {
2       case 4:
3           printf("Aprobaste␣raspando\n");
4           break;
5       case 5:
6           printf("Peor␣es␣nada\n");
7           break;
8       case 6:
9       case 7:
10          printf("Bien!\n");
11          break;
12      case 8:
13      case 9:
14          printf("Muy␣bien!\n");
15          break;
16      case 10:
17          printf("Excelente\n");
18          break;
19      default:
20          printf("Qué␣vergüenza!\n");
21          break;
22  }
```

`nota` es la variable entera a evaluar. Cada **case** evalúa un valor exacto particular. Una vez que se entró por igualdad en un **case** se va a ejecutar todo lo que venga a continuación hasta encontrar un **break** (si no hubiera **break** va a seguir de largo, no importa que se metiera en otros **case**). De forma opcional puede haber un **default** que se ejecutaría sólo si no hubo coincidencia con ninguna de las etiquetas de los `cases`.

Cabe destacar que **switch no** es una iteración, acá la palabra reservada **break** se reutiliza para algo que no tiene nada que ver con la función en ciclos ya vista.

Y remarquemos de nuevo: Los valores de comprobación tienen que ser exactos, no hay forma en el **switch** de decir en un **case** "si está entre tanto o tanto" o enumerar valores. Cada testeo exacto necesita de su **case**.

La instrucción **switch** existe porque es muy útil en algunos contextos de bajo nivel. Dado que la evaluación de la variable se hace una única vez y se chequea por valores exactos, la misma puede ser compilada de forma muy eficiente como una tabla de búsqueda (*lookup table*). Para código de alto nivel suele ser mucho más común utilizar un patrón de **if-else** como se mostró en un ejemplo anterior.

5.11. El operador condicional

Además de los operadores aritméticos, de asignación y booleanos que ya vimos el lenguaje C tiene un operador condicional. Va por enésima vez pero repasemos: Un operador es un tipo de expresión que toma un número de operandos y evalúa a un valor.

El operador condicional de C es el único operador ternario, es decir, que tiene 3 operandos. La sintaxis es `a ? b : c`. Si `a` es verdadero evalúa a `b`, si no evalúa a `c`. Ejemplo: `max = (x > y)? x : y;`, si `x` es mayor que `y` entonces evalúa al valor de `x`, si no al de `y`, en `max` se va a asignar el máximo de los dos. Ejemplo: `volumen = (control < 100)? control : 100;`, si `control` es menor a `100` usa el valor que tenga, si no lo limita a `100`.

Es importante destacar que estamos ante un operador. No una instrucción de control de flujo. Y si queremos recordar qué es un operador... me tomé el trabajo de definirlo hace dos párrafos. El operador condicional **no** reemplaza al **if**, no es intercambiable con él, no nos sirve para o hacer una cosa o hacer otra cosa. El operador, como operador, nos sirve para evaluar a uno de dos valores diferentes en el medio de una expresión.

Por fuera de eso, la sintaxis de `a ? b : c` es confusa de leer cuando uno no está acostumbrado, pero es un operador que permite economizar mucho código cuando se utiliza para evitar escribir estructuras de control de flujo. Por poner un ejemplo:

```
1 printf("Faltan␣%d␣minuto%s␣y␣%d␣segundo%s\n",
2                       minutos, minutos == 1 ? "" : "s", segundos,
                           ↪ segundos == 1 ? "" : "s");
```

Si tuviéramos `minutos = 5, segundos = 1` imprimiría `"Faltan␣5␣minutos␣y␣1␣segundo ↪ \n"`. ¿Es críptico? seguro. Ahora bien, si lo hicieras con **if** necesitarías 8 líneas de código para obtener el mismo resultado.

5.12. goto

La primera regla del club del **goto** es no utilizar **goto**.

Es una instrucción súper práctica para gente que sabe. Ustedes no saben. Está prohibido utilizarlo en el curso.[7]

[7] Pero el día que sepan úsenlo, es muy práctico, para algo está.

Capítulo 6

Arreglos

Con las herramientas vistas hasta el momento podemos declarar cuantas variables como queramos, ahora bien el acceso a cada una de esas variables necesita la escritura de código específico para manipular a cada una de ellas. Imaginemos que tenemos un problema donde tenemos que guardar múltiples valores de una misma especie, por ejemplo, para 50 alumnos en un curso queremos almacenar su nota de parcial. ¿No se haría insostenible mantener variables `nota1`, `nota2`, `nota3` ... `nota50` y escribir código para acceder a cada una de ellas?, ¿y qué haríamos si el cuatrimestre siguiente tuviéramos 60 alumnos en vez de 50?

El lenguaje de programación C permite declarar paquetes de variables del mismo tipo. Estos paquetes se llaman arreglos o arrays o vectores. Un arreglo consiste en un bloque de memoria consecutivo con espacio suficiente como para almacenar n variables de determinado tipo y cada una de esas variables puede ser accedida de forma individual utilizando el nombre del paquete y un número que representa su índice. La línea:

```
1 int valores[4];
```

declara un arreglo de nombre `valores` que contiene 4 elementos cada uno de tipo **`int`**. Dado que no definimos el arreglo, cada uno de estos enteros contendrá basura.

Podemos acceder a cada uno de estos enteros y definirles un valor:

```
1 valores[0] = 10;
2 valores[1] = 20;
3 valores[2] = 30;
4 valores[3] = 40;
```

Dentro de `valores` tenemos 4 variables de tipo **`int`** que están numeradas entre 0 y 3. Todo arreglo de n elementos tiene sus elementos entre las posiciones 0 y $n-1$. Cada uno de los `valores[i]` se comportará como una variable de tipo **`int`** independiente de las demás y con **todas** las reglas que ya conocemos.

El lenguaje C también permite definir nuestro arreglo en bloque el momento (y sólo en el momento) de la declaración:

```
1 int valores[4] = {10, 20, 30, 40};
```

En este caso pedimos memoria para `4` elementos **`int`** inicializados con los valores entre las llaves.

Siendo que estamos inicializando con 4 valores podemos omitir la longitud:

```
1 int valores[] = {10, 20, 30, 40};
```

generará el mismo arreglo que en el ejemplo anterior. El compilador contará cuántos valores se definen y declará al arreglo con ese tamaño.

En caso de dar un tamaño y definir con una cantidad diferente:

```
1 int valores[4] = {10, 20};
```

el compilador declarará un arreglo de 4 enteros y definirá **TODOS** los elementos del arreglo. Los primeros dos, que están especificados, con `10` y `20` y los otros dos con `0`. Si, por ejemplo, quisiéramos inicializar en cero un arreglo de una cantidad arbitraria de elementos podemos utilizar esto a nuestro favor y definirlo como `{0}`, esto forzará la definición del primer elemento... pero eso disparará que se definan todos los restantes.

Es importante destacar que inicializar un arreglo es un proceso costoso que requiere que el compilador itere sobre la memoria. No vamos a inicializar los valores de un arreglo a menos que nuestro problema lo requiera.

Además se pueden inicializar posiciones específicas de un arreglo, por ejemplo:

```
1 int valores[4] = {[3] = 40, [1] = 20, 30};
```

inicializa la posición 3 con `40`, la posición 1 con `20` y la siguiente a 1, es decir la 2, con `30`.

Vamos a focalizar sobre algo que ya se dijo: Cuando declaramos un arreglo generamos un paquete que contiene en su interior *n* variables de un determinado tipo. C no provee ninguna herramienta para, por ejemplo, imprimir un arreglo. ¿Cómo imprimimos entonces un vector?, bueno, como cada uno de los elementos individuales que contiene, que pertenecen a tipos básicos que sabemos operar.

6.1. La memoria de los arreglos

Como se dijo previamente, cuando uno declara un arreglo el compilador reserva un paquete de memoria consecutiva de tamaño suficiente como para contener nuestros elementos. En el ejemplo

```
1 int valores[5];
```

como `sizeof(int)= 4`[1] entonces el compilador reservará 20 bytes de memoria consecutivos, 5 veces los 4 bytes que necesita cada uno de sus enteros.

De esa memoria el compilador sólo recordará que la memoria comienza en una posición determinada, por ejemplo la posición de memoria `0xA4`. Para el compilador, internamente, `valores` va a ser recordado por su posición. Esto no es algo particular de arreglos, ya se mencionó que así funciona siempre que se declara una variable. El nombre de la variable es un identificador recordable para nosotros los programadores, el compilador conoce en qué posición de memoria asignó esa variable.

¿Y cómo accederá el compilador a la memoria de cada uno de los elementos particulares de mi arreglo? Pues haciendo cuentas. El compilador sabe que el primer elemento está, en nuestro ejemplo, en la posición `0xA4`. Si cada elemento mide 4 bytes el segundo elemento estará en la posición `0xA8`, el siguiente en la posición `0xAC`[2] y así (figura 6.1).

Es decir, internamente para nuestro ejemplo cada elemento `i`-ésimo se encuentra en memoria en la posición `0xA4 + i * 4`.

[1] Vamos a recordar por única vez que, salvo para `sizeof(char)` todos los `sizeof` dependen de la plataforma y del compilador pero en este curso vamos a utilizar de ejemplo al GCC en 64 bits. Es la última vez que haremos esta aclaración.

[2] Las direcciones de memoria suelen expresarse en hexadecimal. Como el hexadecimal tiene 16 símbolos, cuando se nos acaban los símbolos del `0` al `9` del decimal continuamos con letras, de la `A` a la `F`. La `C` representa el valor 12, y es lógico que $8 + 4 =$ `C`.

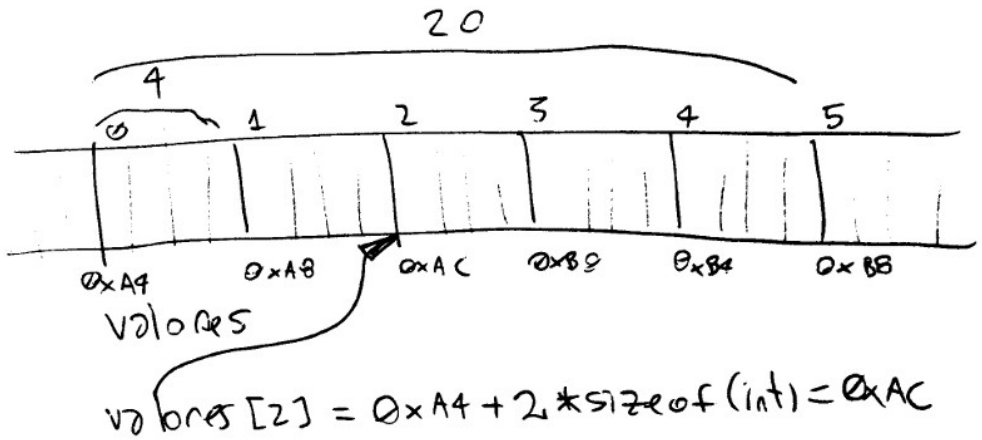

Figura 6.1: Esquema del arreglo valores y del acceso a `valores[2]`.

Debería tener todo el sentido ahora que en C el primer elemento sea el de la posición 0, es simplemente porque es donde arranca la memoria del arreglo.

¿Por qué es importante entender cómo funciona la memoria en los arreglos? Porque la siguiente pregunta que tenemos que hacernos es: ¿Qué hace C si quisiera acceder por encima del índice $n - 1$ en un arreglo de n elementos?

La pregunta se responde con el primer capítulo de este apunte. Cuando en ingeniería diseñamos un producto tenemos objetivos y lineamientos previos al diseño. La realidad es que no hay productos buenos y malos en absoluto, será bueno un producto que cumpla con sus objetivos y malo uno que no. Recordemos entonces para qué se creó C: Se necesitaba un lenguaje que sirviera para escribir sistemas operativos, que las operaciones sean traducibles directamente a assembly, que no haga nada que el programador no pidió explícitamente hacer.

Así como en C la memoria de las variables no se inicializa si no las definimos o así como en C los overflows simplemente ocurren, si queremos acceder por fuera de la memoria de un arreglo nuestro programa intentará hacerlo. Es un comportamiento totalmente deseable... no deseable para un programador, pero sí deseable según las premisas de diseño y según lo que espera alguien que va a programar economizando recursos.

Si en nuestro ejemplo anterior accediéramos a `valores[5]` el compilador hará la cuenta `0xA4 + 5 * 4 = 0xB4` y accederá a esa posición de memoria. Que está fuera de la memoria de nuestro vector. Es un error **gravísimo**, es un error que puede hacer que el sistema operativo mate a nuestro programa, o puede hacer que nuestro programa tenga un comportamiento errático o incluso abrir un bache de seguridad importante, pero no es un error del lenguaje, es un error del programador.

6.2. El tipo `size_t`

No hay dos arquitecturas de procesador iguales, y esto aplica también a cuánta memoria puede administrar determinado procesador o plataforma. Por ejemplo el procesador MOS6502, que es el procesador que estaba en todos los videojuegos y consolas de los '80s y principios de los '90s, era un procesador de 8 bits, sin embargo cuando se trataba de memoria RAM podía manejar direcciones de memoria de 16 bits[3]. Mientras que las PCs generalmente pueden indexar tanta memoria como la capacidad del procesador, por ejemplo una Intel 80286 de 16

[3]Y si no lo hubiera hecho no hubiera podido usar más de 256 bytes de memoria, con 16 bits eso se expande a 65536. Recordemos que la memoria no sólo sirve para almacenar variables, también es donde se encuentra el código máquina de nuestro programa.

bits indexaba 16 bits, la Pentium III de 32 utilizaba 32 y un procesador actual de 64 indexa 64.

La pregunta que cabe hacerse es, si tuviera que almacenar una cantidad referida a tamaños de memoria, ¿cuál sería la mejor variable para hacerlo?, ¿**short**, **int**, **long**? Ya que estamos un comentario al margen, históricamente el tamaño de palabra del procesador era lo que se utilizaba para las variables **int**, sin embargo ya vimos que en el GCC de 64 bits en vez de tener **int** de 64 se tienen de 32... lo cual incluso subvierte hasta poder adivinar la capacidad de memoria en función del tipo **int**.

La pregunta que hicimos en el párrafo anterior no tiene respuesta satisfactoria. La única respuesta posible sería "usá el tipo más grande y eso te va a dejar seguro", pero tal vez el tipo más grande sea ineficiente tanto en memoria como en operatoria porque si el hardware no lo soportara se requeriría operarlo por software que es groseramente más lento.

Para responder a esta pregunta tenemos el tipo `size_t`. El tipo `size_t` es un tipo provisto por el compilador. Como el fabricante del compilador sabe en qué plataforma está y qué decisiones de mapeo de tipos utilizó sabe cuál es el tipo adecuado para guardar una cantidad de memoria que va a funcionar correctamente en esa plataforma. Por lo tanto se tomó el trabajo de hacer un **typedef** para `size_t` con el tamaño adecuado.

En C `size_t` es el tipo de los índices de los vectores, también es el tipo de lo que devuelve **sizeof**.

Es tan generalizado el uso de `size_t` y tan opaco qué tipo tiene detrás que incluso hay definido un formato para imprimir variables de tipo `size_t` en `printf()`:

```
1 printf("%zd\n", sizeof(int)); // Imprimiría 4
```

En este curso vamos a utilizar `size_t` **siempre** que manipulemos cantidades de memoria o índices de arreglos.

6.3. El problema del `sizeof` de los arreglos

Supongamos el siguiente ejemplo:

```
1  #include <stdio.h>
2
3  void f(int valores[]) {
4      printf("%p\n", valores);
5      printf("%zd\n", sizeof(valores));
6  }
7
8  int main(void) {
9      int valores[] = {1, 2, 3, 4, 5};
10
11     printf("%p\n", valores);
12     printf("%zd\n", sizeof(valores));
13
14     f(valores);
15
16     return 0;
17 }
```

y volvamos a asumir que `valores` vive en la posición `0xA4` de memoria. Al ejecutar el programa los `printf()` del `main()` imprimirían `0xA4` y `20` respectivamente. No hay mucho más que decir, por un lado `"%p"` es el modificador que utilizamos para imprimir posiciones de memoria y por fuera de eso ambos valores son lo esperado.

Ahora bien, dentro de la función `f()` se imprimirán `0xA4` y `8` respectivamente. ¿Qué? Eso. No parece tener sentido, ¿no?, ¿la variable `valores` que pertenece a `f()` contiene la misma posición de memoria que la variable `valores` que pertenece al `main()`? ¿Por qué 8 si el vector tiene 5 enteros y debería totalizar 20?

En este capítulo no responderemos ninguna de estas preguntas. Sólo presentamos el problema. No importa el tamaño del vector el **`sizeof`** dentro de una función será siempre 8 y hay algo en cómo funciona C que hace que, para arreglos, la variable dentro de la función copie la dirección de la variable fuera.

Notar que dentro de la función que declaró un arreglo podríamos utilizar **`sizeof`** para saber la cantidad de elementos del mismo:

```
1 void f(void) {
2     double valores[] = {1, 2, 3, 4, 5};
3     size_t n = sizeof(valores) / sizeof(valores[0]);     // Da 5
4 }
```

No importa el tamaño del tipo de los elementos, si dividimos el tamaño de la memoria total por el tamaño de uno de los elementos obtendremos el número de elementos. **Sólo** dentro de la función que lo declara, como vimos antes.

Esto que vimos referido a las funciones tiene consecuencias importantes en la práctica. Supongamos que queremos implementar una función que reciba un arreglo de enteros y devuelva el promedio de sus elementos. La firma no podría ser **`float`** `promediar(`**`int`** `a[]);` a menos que haya una forma externa de conocer la longitud del arreglo `a`, porque el problema me lo defina de alguna forma. De forma genérica siempre deberá recibir el tamaño:

```
1 float promediar(int a[], size_t n) {
2     int suma = 0;
3     for(size_t i = 0; i < n; i++)
4         suma += a[i];
5 
6     return (float)suma / n;
7 }
```

Notar que tanto para el tamaño como para el índice de iteración utilizamos `size_t`. Notar además que está función es genérica con respecto al tamaño del arreglo, funciona para arreglos de cualquier tamaño. Cuando nosotros escribimos una función de bajo nivel no conocemos cuál es el contexto en el cual la misma va a ser utilizada. Siempre vamos a elegir diseñar funciones genéricas que se adapten a cualquier tamaño de problema.

6.4. Arreglos multidimensionales

En C podemos declarar arreglos de más de una dimensión, es decir arreglos de arreglos. Es lo que vamos a utilizar, en principio, para modelar matrices.

La expresión:

```
1 int m[4][3] = {
2     {11, 12, 13},
3     {21, 22, 23},
4     {31, 32, 33},
5     {41, 42, 43},
6 };
```

declara una matriz de 4 filas y 3 columnas que en Álgebra podríamos pensar como:

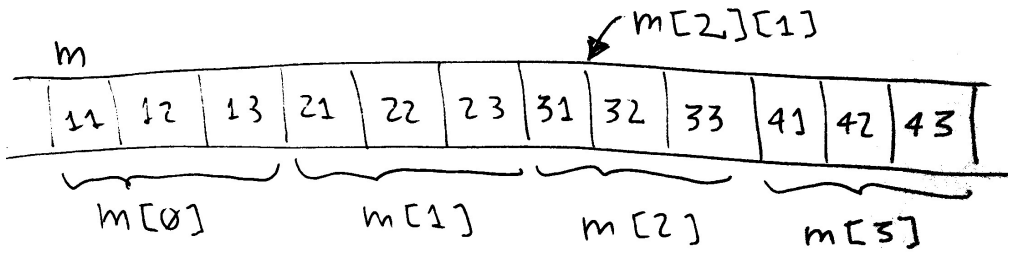

Figura 6.2: Esquema de la matriz `m` y elemento `m[2][1]`.

$$M = \begin{pmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \\ 41 & 42 & 43 \end{pmatrix}.$$

Podemos acceder a cualquier elemento de la matriz utilizando dos índices, por ejemplo `m[2][1]` es el elemento de valor `32` dado que se trata de la tercera fila y segunda columna.

Internamente la memoria se agrupa de forma lineal, una fila a continuación de la otra: `{11, 12, 13, 21, 22, 23, 31, 32, 33, 41, 42, 43}`[4]. ¿Cómo hace el compilador para darnos la ilusión de que la memoria es bidimensional?

C en realidad ve a `m` como un arreglo de 4 elementos donde cada elemento es un arreglo de 3 enteros. Por eso empezamos diciendo que los arreglos multidimensionales son arreglos de arreglos. Cuando el compilador procesa la expresión `m[2][1]` primero ataca `m[2]`. Si `m` es un arreglo de arreglos, entonces `m[2]` es el tercero de esos arreglos y es el que declaramos como `{31, 32, 33}`. ¿Cómo llega a eso?, haciendo la misma cuenta que ya desarrollamos para arreglos unidimensionales. Si cada elemento de `m` es un arreglo de 3 enteros entonces el **`sizeof`** de cada elemento es $3 \times 4 = 12$. Cuando decimos `m[2]` entonces va a saltearse `2 * 12` desde el inicio y va a ir al byte 24, es decir, 6 enteros más adelante que el comienzo de la memoria. Es decir, va a estar sobre la memoria que se corresponde con el valor `31`. Luego aplica `[1]` sobre ese arreglo `{31, 32, 33}` es decir, se saltea 4 bytes. Ahí obtiene que `m[2][1]` se corresponde con el `32` (figura 6.2).

Notar que genéricamente para `m[i][j]` el compilador se mueve `i * 3 + j` unidades de enteros desde el inicio de la memoria.

6.4.1. Pasaje de matrices a funciones

El pasaje de matrices a funciones es otra de las cosas que vino a solucionar el estándar C99 y que en los 30 años previos no funcionaba de un modo natural.

Supongamos que tenemos definido:

```
1 int m[2][3] = {
2     {0, 1, 2},
3     {3, 4, 5},
4 };
```

y queremos pasarle esa matriz a una función que incialice en 0 cada uno de sus elementos.

En las variantes de C previas a C99 deberíamos haber implementado algo de este estilo:

```
1 void funcion1(int m[2][3]) {
2    for(size_t f = 0; f < 2; f++)
3       for(size_t c = 0; c < 3; c++)
4          m[f][c] = 0;
```

[4]Ordenar primero por filas es lo que se conoce como *row-major order*, no todos los lenguajes usan esta convención.

```
5 }
```

e invocado `funcion1(m);`. Notar que es lo que dijimos que no queríamos, nosotros queremos tener funciones genéricas que operen con arreglos (y matrices) de cualquier tamaño. ¿Por qué una función que opera sólo con matrices de 2×3 si podríamos hacerlo con matrices de tamaño arbitrario.

Bueno, previo a C99 lo más "genérico" que podíamos obtener era esto:

```
1 void funcion2(int m[][3], size_t fs) {
2    for(size_t f = 0; f < fs; f++)
3       for(size_t c = 0; c < 3; c++)
4          m[f][c] = 0;
5 }
```

e invocado `funcion2(m, 2);`. En el pasaje de matrices a funciones puede omitirse la primera dimensión como parte del tipo. ¿Sabemos explicar por qué la primera dimensión no importa pero sí las demás? Sí, claro, es lo que vimos en el capítulo anterior. Dentro de las dos funciones que definimos hasta ahora cuando se accede a `m[f][c]` el compilador tiene que hacer una cuenta que involucra moverse `f * 3 + c` enteros con respecto al comienzo de la matriz[5]. Notar que en esa cuenta importa la cantidad de columnas pero la cantidad de filas nos tiene sin cuidado. C necesita tener definido ese número para poder computar el acceso a la memoria.

Seguimos sin tener funciones genéricas.

En el estándar C99 se introdujo una sintaxis nueva que es la siguiente:

```
1 void funcion3(size_t fs, size_t cs, int m[fs][cs]) {
2    for(size_t f = 0; f < fs; f++)
3       for(size_t c = 0; c < cs; c++)
4          m[f][c] = 0;
5 }
```

que invocaríamos `funcion3(2, 3, m);`. Finalmente tenemos funciones para matrices genéricas, podemos pasarle matrices de cualquier dimensión. **Pero** estamos **obligados** a que las dimensiones estén en la firma de la función **antes** de la matriz. Notar que no es que no especificamos el tamaño de `m`, estamos diciendo que `m` tiene `cs` columnas (podríamos omitir las filas) por lo tanto el parámetro `cs` tiene que venir antes en la definición de `funcion3()`.

Dado que en este curso utilizamos el estándar C99 preferiremos la variante de `funcion3()`, incluímos las demás por contexto histórico.

6.5. Arreglos de largo variable (VLA)

En el estándar C99 se introdujeron los arreglos de largo variable (VLA por las siglas de *variable length array*). En un VLA el tamaño del arreglo puede ser definido por una variable:

```
1 size_t n;
2 // Asigno un valor para n
3 int v[n];
```

En versiones previas del lenguaje los tamaños de los vectores tenían que ser números fijos definidos en tiempo de compilación, dado que la asignación de la memoria se hacía de forma estática.

En el ejemplo anterior, el vector `v` se crea con el tamaño que tenga el valor de `n` en el momento de la declaración. Una vez que `v` está creado su tamaño ya no cambia.

[5] Que por lo que vimos en el capítulo de arreglos y **`sizeof`** está codificado en `m`.

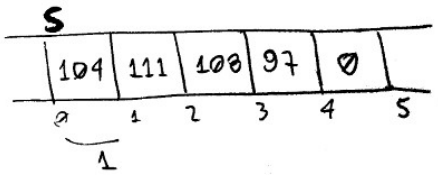

Figura 6.3: Esquema del arreglo que contiene la cadena `"hola"`.

Si bien los VLA son una herramienta útil del lenguaje, su uso debe ser evitado. En próximos capítulos desarrollaremos más sobre cómo funciona la memoria y veremos que el tamaño máximo de los vectores que estuvimos manipulando hasta el momento es muy limitado. Querer crear un vector con un largo grande va a hacer que nuestra aplicación se rompa, por lo que los VLA no son la herramienta para generar vectores de tamaño adaptativo a los problemas. Sólo nos van a servir si el tamaño del vector está acotado. Más adelante en el curso veremos técnicas para resolver este tipo de problemas.

6.6. Cadenas de caracteres

Venimos utilizando cadenas de caracteres desde el hola mundo, son precísamente el `"Hola␣mundo\n"` que le pasamos como parámetro a `printf()`. Las cadenas de caracteres son un caso particular de arreglos de **`char`** con un par de reglas adicionales propias.

En primer lugar repasemos que un carácter se almacena en memoria como un número. La equivalencia entre números y caracteres que solemos utilizar es la dada por la tabla ASCII[6]. Dicha tabla define 127 caracteres que podemos identificar con su respectivo número el cual puede ser almacenado sin problemas en un byte.

Bien, volvamos ahora sobre el hola mundo:

```
1 printf("Hola␣mundo\n");
```

Si las cadenas de caracteres son arreglos de caracteres, entonces a `printf()` le estamos pasando un arreglo... ¿Pero no era que si pasábamos un arreglo a una función teníamos que además pasar su longitud? Sí, siempre y cuando no haya información adicional para inferir esa longitud.

Las tres instrucciones siguientes son totalmente equivalentes entre sí:

```
1 char s[] = "hola";
2 char s[] = {104, 111, 108, 97, 0};
3 char s[] = {'h', 'o', 'l', 'a', '\0'};
```

Es decir, las 3 líneas declaran un arreglo de 5 caracteres y lo definen con los ASCIIs correspondientes a la h, la o, la l, la a... y el carácter 0 de la tabla ASCII, llamado NUL y representado en C con el literal `'\0'` (figura 6.3).

Si bien cuando se desarrolló C el diseño de sus cadenas de caracteres fue novedoso no hay magia negra en el hola mundo: `printf()` sabe la longitud de la cadena que recibe porque la misma está delimitada. Su último carácter es un centinela que dice que no hay que seguir más allá de él. En realidad no sabe la longitud, pero puede calcularla sencillamente iterando cada uno de los caracteres.

Notar una sutileza, el párrafo anterior dice *"sabe la longitud de la cadena que recibe"* y no *"sabe la longitud del arreglo que recibe"*. Arreglo y cadena son dos cosas diferentes:

[6] ASCII: American Standard Code for Information Interchange, es decir código "americano" estándar para el intercambio de información.

```
1 char s[100] = "hola";
```

En este ejemplo tenemos un arreglo `s` de longitud 100, ahora bien, la cadena contenida en dicho arreglo mide 4: h, o, l, a. La relación entre arreglo y cadena es que la longitud del arreglo tiene que ser al menos uno más que la longitud de la cadena, para poder almacenar el `'\0'`. Pero arreglo y cadena son cosas diferentes. El arreglo es el contenedor en el cual vive la cadena.

Sólo por formalizar vamos a definir a la cadena como una sucesión de caracteres en memoria finalizados con un `'\0'`. Y vale destacar que esto es un caso particular de arreglos y más aún un caso particular de arreglos de caracteres.

6.7. Encabezado `string.h`

TODO

6.8. Entrada y salida (I/O)

La comunicación de nuestro programa con el exterior se da en principio por cadenas de caracteres[7]. Desde el primer ejemplo vimos que podemos imprimir una cadena con `printf()`. Ahora vamos a profundizar un poco en esa comunicación.

Un programa en C se comunica con el exterior mediante tres flujos (*streams*) diferentes: `stdin`, `stdout`, `stderr`. El primero es un flujo de entrada, los otros dos son flujos de salida.

¿Y qué son los flujos?, los flujos son colas de comunicación en los cuales se acumulan caracteres. Cuando nosotros imprimimos con `printf()` estamos escupiendo caracteres de a uno por vez a `stdout`. `stdout` va a acumular esos caracteres en una memoria intermedia, llamada *buffer*, y cuando sea conveniente esos datos se van a mostrar por la pantalla u otro lugar equivalente. Se llaman flujos por eso, son un caudal de información que viaja en un determinado sentido, no hay ningún tipo de orden superior a ese continuo de bytes.

Como se dijo, los flujos se separan entre flujos de entrada y de salida. En los de entrada hay algún agente externo (por ejemplo un usuario tipeando en su teclado) que está dejando cosas dentro de un buffer que van a almacenarse ahí hasta que nosotros nos decidamos a leer caracteres de a uno por vez. Mientras que en los flujos de salida nosotros depositaremos caracteres en un buffer, hasta que los mismos se liberen al exterior (por ejemplo al monitor del usuario).

6.8.1. Salida

Si quisiéramos imprimir un byte en `stdout` podemos utilizar:

```
1 putchar('h');
```

imprime la letra `'h'` en `stdout`.

Tenemos dos funciones que trabajan con cadenas:

```
1 puts("hola");
2 printf("%d\n", 42);
```

La primera imprime `"hola\n"` por `stdout`. Prestar atención al `'\n'` que agregó aunque nosotros no lo incluímos. La segunda es una vieja conocida e imprime `"42\n"`. Cuando decimos "imprime" en todos los casos estamos queriendo decir: Escupe esos bytes en el buffer de `stdout` y se desentiende por lo que pase después. No es mi programa el que pone las cosas en la pantalla, si es que hay una.

[7] Y por el entero que devuelve el `main()`, pero no podemos expresar mucho por ahí.

6.8.2. Entrada

Si quisiéramos leer un único carácter, por ejemplo ingresado por el usuario, de `stdin` podemos utilizar:

```
1 int c = getchar();
```

en la variable `c` quedará el byte leído. Si no hubiera nada en el buffer de `stdin` mi programa se quedará esperando a que hayan datos para leer. Cabe destacar que los datos generalmente se vuelcan al buffer recién después de que el usuario presione el "enter", el cual representa uno o dos caracteres y es lo que nosotros vemos como `'\n'`.

También podríamos leer una línea completa, hasta el `'\n'` (inclusive):

```
1 char s[30];
2 fgets(s, 30, stdin);
```

La función `fgets()` intenta leer una línea de `stdin`. Nosotros le indicamos el tamaño de memoria que tenemos disponible, en este caso 30, por lo que la función sabe que tiene hasta 29 caracteres disponibles, dado que para generar una cadena hay que finalizar con `'\0'`. Como lee líneas si el `'\n'` se alcanzan antes de agotar los 29 caracteres la función leerá hasta el `'\n'` (y si hubieran más cosas en el buffer quedarán esperando a una siguiente lectura) y finalizará la cadena. En cambio si se agotaran los 29 caracteres y no se hubiera leído el `'\n'` la función finalizará la cadena y retornará, es decir, sin haber leído la línea en su totalidad. Mirar si el último carácter de la cadena es un `'\n'` nos permite saber si el tamaño de nuestro arreglo fue suficiente o la línea lo superó.

6.8.3. Leer cosas que no son cadenas

No se puede, `stdin` es un stream de caracteres.

Lo que sí se puede es leer una cadena y luego procesarla para extraer lo que necesitemos.

Por ejemplo, si necesitáramos leer un entero podríamos hacer:

```
1 char s[20];
2 fgets(s, 20, stdin);
3 int n = atoi(s);
```

La función `atoi()`, declarada en el encabezado `<stdlib.h>` recibe una cadena de caracteres e intenta extraer el valor numérico contenido en la misma. Por ejemplo, si se invocara `atoi("` ↪ `130");` la función devolvería el entero `130`. La función convierte a valor numérico mientras encuentre en la cadena caracteres válidos, cuando encuentre un caracter no numérico interrumpirá el proceso y devolverá lo que tenía hasta ese momento. Por ejemplo, `atoi("130hola");` devolverá `130`, dado que la `'h'` no forma parte de un número.

Análoga a la función `atoi()` está la función `atof()` que sirve para extraer un número **`float`** de una cadena.

6.8.4. Agotando la entrada

Vimos que con `getchar()` podemos leer un byte y que con `fgets()` podemos leer una línea. Ahora bien, ¿cómo haríamos si quisiéramos leer todos los bytes o todas las líneas hasta agotar la entrada?

Tenemos los siguientes códigos:

```
1 int main(void) {
2     int c;
3     while((c = getchar()) != EOF)
```

```
4           putchar(c);
5       return 0;
6   }
```

y

```
1   int main(void) {
2       char s[100];
3       while(fgets(s, 100, stdin) != NULL)
4           printf("%s", s);
5       return 0;
6   }
```

Ambos códigos van a leer la totalidad de lo que haya para leer, en un caso de a un carácter por vez y en el otro de a líneas (o fracciones de líneas, si alguna de ellas midiera más de 99 caracteres) y hacer un eco de lo leído por `stdout`.

La función `getchar()` devuelve `EOF` en caso de *falla* mientras que `fgets()` devuelve `NULL`, ambas son etiquetas. ¿Qué se considera una falla? Que se termine la entrada, EOF literalmente son las iniciales de *end of file*, final de archivo. Esta marca se dispara cuando ya no hay nada para leer.

¿Cómo hace el usuario para decirle a mi programa que no va a ingresar más datos? La señal de final de archivo se dispara con la combinación de teclas: Control + D, apretando ambas teclas a la vez.

Al principio dijimos que los caracteres se codificaban en un byte... ¿por qué entonces `getchar()` devuelve un **`int`** y no un **`char`**? Bueno, el valor de `EOF` no es un **`char`**. La función devuelve **o** caracteres **o** `EOF`. Entonces no le alcanza un **`char`** para su devolución. El valor leído de `getchar()` nosotros tenemos que almacenarlo en un **`int`**, pero una vez que validamos que ese **`int`** no sea `EOF` entonces sabemos que entra en un **`char`**:

```
1   char s[100];
2   size_t i;
3   int c;
4
5   while(i < 99 && (c = getchar()) != EOF && c != '\n')
6       s[i++] = c;
7   s[i] = '\0';
```

Sería un algoritmo similar al que implementa internamente `fgets()`.

6.8.5. Redirección de flujos

Si bien se dijo que generalmente `stdin` es un flujo que representa lo que ingresa por teclado y `stdout` lo que sale por la pantalla, este comportamiento puede cambiarse fácilmente a nivel sistema operativo:

```
$ ./programa > archivo.txt
```

Ejecuta a `programa` y redirige su salida de `stdout` al archivo `archivo.txt` (si no existe lo crea, si existe lo reemplaza).

Análogamente:

```
$ ./programa < archivo.txt
```

Ejecuta a `programa` pasándole el contenido completo del archivo por `stdin`. Al terminar el archivo se dispara, obviamente, la marca de fin de archivo.

También se pueden encadenar programas:

```
$ ./programa1 | ./programa2
```

Ejecuta ambos programas y vuelca el `stdout` del `programa1` como `stdin` del `programa2`. Esta técnica de redirigir "streams" se conoce como "piping", literalmente entubado. De ahí el carácter | toma el nombre de "pipe".

6.8.6. Salida de error

Como se dijo nuestro programa tiene dos flujos de salida `stdout` y `stderr`. El primero corresponde a la salida normal de nuestro programa, lo que se supone que es parte del procesamiento que realiza. El segundo se utiliza para informar de errores anormales en el programa.

Para imprimir por `stderr` usamos la función:

```
1 fprintf(stderr, "Error: %d\n", 10);
```

que es similar a `printf()` pero se antepone un parámetro adicional que es el flujo de salida en el que queremos escribir[8].

Dos cosas importantísimas sobre `stderr`:

1. `stderr` **no** tiene buffer, por lo que todo lo que imprimamos se va a ver de forma **inmediata**. Cuando imprimimos por `stdout` dado que hay un buffer, si nuestro programa se rompiera en la línea siguiente tal vez lo último que imprimimos no llega a verse.

2. La salida de `stderr` **no** es capturada cuando hacemos redirección, es decir no importa que hagamos piping **siempre** se van a ver los errores por pantalla.

Es por esto que utilizaremos `stderr` siempre que queramos mostrarle errores al usuario y también siempre que estemos "poniendo `printf()`s" para diagnosticar un programa que falla.

[8] En realidad es mucho más que similar... cuando invocamos a `printf(...);` esta función no hace otra cosa que invocar a `fprintf(stdout, ...);`.

Capítulo 7

Alcance de variables

Hasta ahora hablamos de variables pero no profundizamos en dónde se ubican esas variables en la memoria ni qué visibilidad tienen.

7.1. Globales y locales

En el lenguaje C hay dos tipos de variables: globales y locales.

Las variables globales son aquellas declaradas fuera de las funciones, mientras que las locales son las declaradas dentro de las funciones.

Como su nombre lo indica la visibilidad de las variables globales es desde todos lados, mientras que la visibilidad de las variables locales es sólo dentro de la función que las define.

En el siguiente ejemplo:

```
1  #include <stdio.h>
2
3  char a = 'A';
4  char b = 'B';
5
6  void f(char b) {
7      a = 'a';
8      b = 'x';
9  }
10
11 int main(void) {
12     char b = 'b';
13     printf("%c␣%c\n", a, b);
14     f(b);
15     printf("%c␣%c␣%c\n", a, b);
16
17     return 0;
18 }
```

La salida producida es `"A␣b\n"`, `"a␣b"`. ¿Por qué?

Separemos las variables según su pertenencia:

Globales: `a` y `b`

Locales a `main()`: `b`

Locales a `f()`: `b`

Esas 4 variables ocupan lugares diferentes de memoria y son independientes entre sí.

Cuando dentro de un código se utiliza un identificador tienen precedencia las variables locales por sobre las globales. En este ejemplo, en ambas funciones la variable global `b` es invisible dado que hay variables locales con el mismo nombre. Toda modificación que se haga sobre `b` será local a esa función.

En el primer `printf()` se imprime la variable `a` global y la variable `b` local a `main()`.

Luego se llama a la función `f()` la cual redefine la variable global `a` y la variable local `b`.

En el segundo `printf()` se imprimen las mismas variables que antes, pero como `f()` modificó la variable global `a` se imprime el nuevo valor.

7.2. La pila de ejecución

Antes que nada recordemos que las variables viven en la memoria y que a la memoria se accede a través de posiciones de memoria. Nosotros en nuestra aplicación identificamos a las variables por un identificador y es el compilador el que traduce ese nombre en una dirección de memoria.

Las variables globales viven en una zona específica de la memoria de mi programa. Existen durante toda la ejecución y están siempre en la misma posición. Es decir, en nuestro ejemplo anterior, el compilador sabe la ubicación de las variables globales `a` y `b` y si tuviera que referir a ellas en algún lugar reemplazaría su nombre por su posición.

En cambio las variables locales solamente existen mientras la función a la que pertenecen está en ejecución. Por ejemplo, las variables locales de `f()` existen sólo cuando alguien llama a esa función, al igual que las de `printf()`. Luego de ejecutarse las funciones esta memoria queda disponible para reutilizarse.

La memoria de las variables locales vive en una zona que se denomina *stack* o pila. En matemáticas una pila es una estructura donde cada vez que agrego un elemento lo hago encima de los ya existentes, como si se tratara de una pila de platos en una alacena. En nuestro caso lo que se apila es el marco de memoria de cada una de las funciones. Se llama *stack frame*, marco, al espacio que ocupan las variables locales de determinada función.

Si tuviéramos el siguiente código:

```
1  void f() {
2      int a = -3;
3  }
4
5  void g() {
6      int a = -2;
7      f();
8  }
9
10 int main() {
11     int a = -1;
12     f();
13     g();
14     return 0;
15 }
```

Cuando el programa se inicia se ejecuta la función `main()`. La pila estaba vacía, pero se apila sobre ella el marco de la función `main()`. En dicho marco de memoria hay un par de valores que ahora no importan y está la memoria de la variable `a`.

Ahora bien, a diferencia de lo que dijimos con las globales, en el caso de las variables locales el compilador no conoce su posición. La manera en la que identifica a las variables locales es

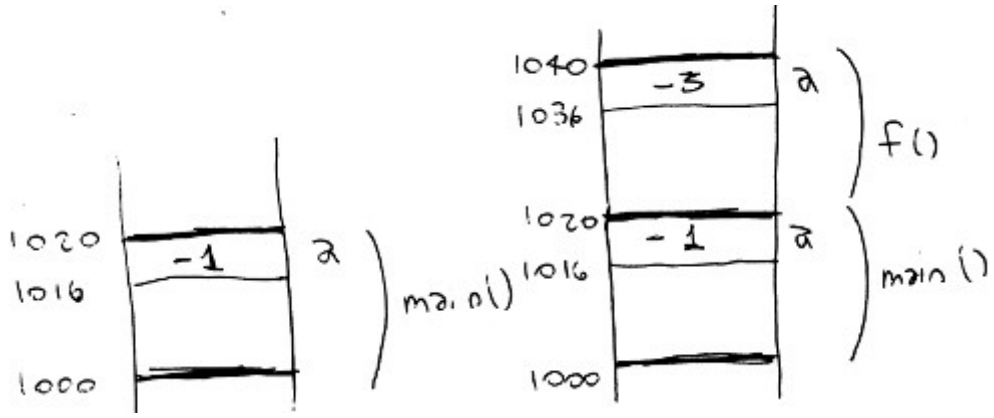

Figura 7.1: Aproximación inicial al esquema de la pila. Izq. La pila al comienzo de la ejecución. Der. La pila al llamar a `f()`.

por su posición relativa dentro del marco. Es decir, el compilador conoce dónde empieza el marco de `main()` y sabe en qué posición dentro de ese marco están las variables locales de esa función.

Supongamos que la variable `a` se encuentra a 16 bytes del inicio del marco de ejecución y supongamos que la pila empieza en la posición 1000 (y si bien la memoria solemos expresarla en hexadecimal vamos a utilizar decimal para facilitar la lectura). Como la variable `a` es entera ocupará 4 bytes, por lo que el marco de `main()` mide 20 bytes (figura 7.1.izq).

Si el marco arrancó en la posición 1000 y la variable `a` está a 16 de distancia, la variable `a` está efectivamente en la posición 1016.

Cuando se llama a la función `f()` se apila sobre el marco de `main()` el marco de `f()` (figura 7.1.der).

Dado que `f()` contiene las mismas variables que `main()` (y que `g()`) su marco de ejecución tendrá la misma disposición.

Ahora bien, el marco de `f()` está encima del marco de `main()`, por lo que empieza en la posición 1020. Por lo tanto la variable `a` de `f()` estará en la posición 1036.

Cuando termina la ejecución de la función `f()`, ¿cómo hace el programa para retomar no sólo la memoria de `main()` si no también el bloque de código máquina que estaba ejecutando antes de ir a `f()`? Acá hay algo importante, la función `f()` puede haber sido llamada desde cualquier lugar, en nuestro caso, estamos siguiendo el código y sabemos que fue llamada desde el `main()` pero podría haber sido llamada desde `g()` (como de hecho pasa más adelante en el código). ¿Cómo resuelve eso mi programa?

Llegó el momento de explicar para qué reservamos 16 bytes en el marco de las funciones. En nuestra plataforma las direcciones de memoria ocupan 8 bytes. Reservamos memoria para guardar dos direcciones de memoria (con significados totalmente diferentes) en el stack.

Volvamos un poco atrás en el tiempo, a antes de que `main()` llamara a la función `f()`. Se está ejecutando la función `main()`, el compilador tiene en la memoria del procesador un registro que le dice dónde empieza el marco de `main()`. Este registro se llama *stack pointer*, puntero de la pila, SP. Además tiene otro registro que se llama *program counter*, contador de programa, PC que le dice en qué posición de la memoria está la instrucción de código máquina que se está ejecutando en ese mismo instante. Al momento de llamar a `f()` el compilador guarda en el marco de memoria de `f()` el valor del SP (que dijimos que valía 1000) y del PC (que podríamos decir que vale 12, el número de la línea que se estaba ejecutando[1]), antes de actualizarlos: Al SP le suma 20 y el PC se reemplaza por donde sea que está el código máquina de `f()` en la

[1]En la vida real va a ser la posición de memoria en la cual está la instrucción de assembly que se está ejecutando, pero queremos que el ejemplo sea seguible.

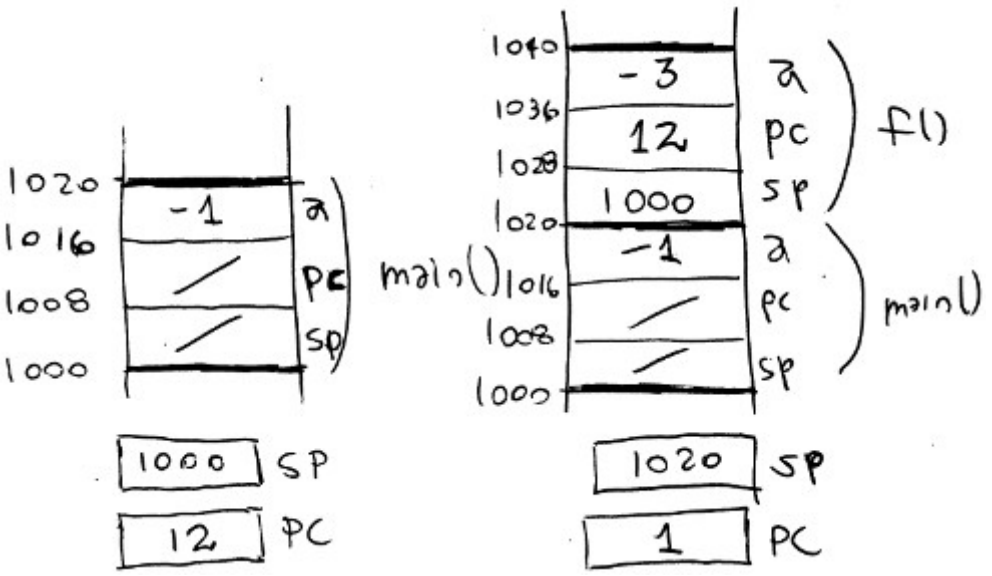

Figura 7.2: Esquema más detallado de la pila y los registros. Izq. La pila justo antes de llamar a `f()`. Der. La pila al llamar a `f()`.

memoria (y, siguiendo nuestra simplificación, diremos que es 1, porque se ejecuta la línea 1 del código fuente). En la figura 7.2.izq se ve cómo era el marco de ejecución en `main()` al ejecutar la línea 12 justo antes de llmar a `f()` mientras que en la figura 7.2.der se ve el instante posterior ya dentro de `f()`, habiendo salvado en la pila de `f()` los valores de reposición de SP y PC previos a la llamada.

Entonces `f()` se ejecuta sabiendo que el marco empieza en lo que indica el SP que es 1020.

Cuando `f()` termina su ejecución tiene que retornar el control a la función invocante. Sabe que al comienzo de su marco están los valores viejos del SP y del PC y los utiliza para restaurar ambos contadores. Eso devuelve el marco al marco de `main()` y la ejecución adonde se había quedado. La memoria de `f()` sigue estando en el stack, pero ya no le pertenece a `f()`, decimos que se destruyó.

Luego `main()` continúa su ejecución y llama a `g()`. Ya sabemos qué hace, apila el SP y el PC en el marco de `g()` e incrementa el SP en 20 y cambia el PC adonde esté el código de `g()`.

La función `g()` ahora tiene su variable local `a` en **la misma** posición en la que antes estuvo la variable local `a` de la función `f()`, dado que el SP está en 20 y la distancia era de 16, en la posición 36 se encuentra `a`.

La función `g()` llama a la función `f()`. Guarda en el marco de `f()` su SP, que vale 1020, y su PC, que vale 7. Incrementa SP en 20, que era el tamaño de su propio marco, y reemplaza el PC por donde esté el código de `f()`, es decir 1. El esquema de la memoria al llamar a `f()` desde `g()` se puede ver en la figura 7.3.

Ahora se ejecuta `f()`, pero notar que la variable `a` de `f()` está en una nueva posición de memoria. Sigue estando a 16 bytes del inicio del stack frame de `f()` pero ahora el SP vale 1040, por lo que `a` está en la posición 1056. La posición de `a` va a depender del historial de llamadas que haya apilados en el stack y del tamaño del marco de cada una de ellas. El compilador la encuentra relativa a lo que le diga el SP, pero el SP es una variable más del problema. En nuestras tres funciones la variable `a` está siempre en SP + 16, y sin embargo es una posición diferente para cada función e incluso para cada llamada a ellas.

Este mecanismo permite que las funciones se llamen libremente, en cualquier orden, que sólo se ocupe memoria para las funciones que están en ejecución en un determinado momento, e incluso permite que la misma función pueda ser llamada más de una vez dado que cada

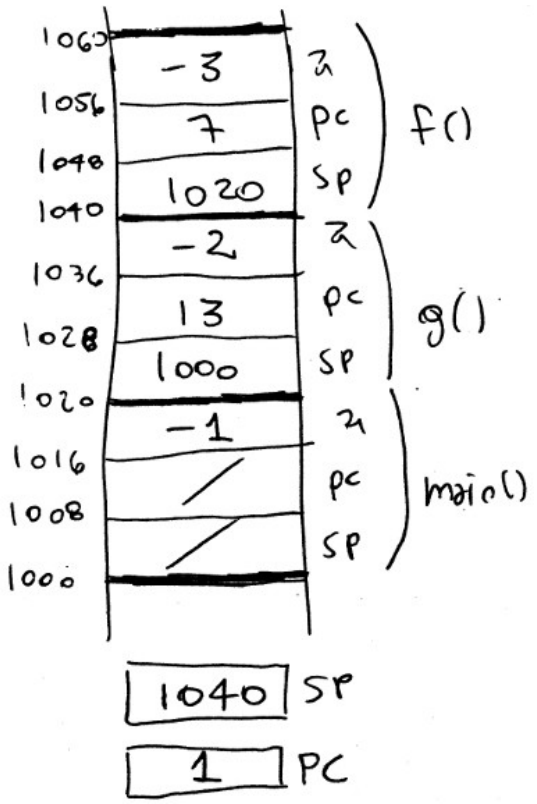

Figura 7.3: Estado de la pila al llamar a `f()` desde `g()`.

invocación tendrá su propio marco de memoria independiente del marco de las otras llamadas. Por el contrario, las variables globales están en una posición fija conocida por todos de forma absoluta.

Como siempre, esto que se contó en estos párrafos es una simplificación del problema para explicar el mecanismo y el concepto. La implementación real de esto tiene muchos más detalles y complejidades[2].

7.3. Paradigma procedural

El paradigma de programación procedural (o procedimental) propone que estructuremos nuestro código en funciones.

Las funciones son cajas negras que resuelven subproblemas de mi problema original y la comunicación de las funciones con sus funciones invocantes es a través de los parámetros de la función y el retorno de la misma. Es decir, si quiero invocar a una función `f()` le tendré que pasar parámetros con la entrada que necesita dicha función y la función me devolverá con un **`return`** el resultado de su ejecución.

Los parámetros de entrada y valores de retorno de la llamada a la función tienen que ser explícitos.

Esto quiere decir que el ejemplo que vimos cuando hablamos de variables globales donde imprimíamos unas variables, llamábamos a una función **`void`**, reimprimíamos las mismas variables y el resultado era diferente es inaceptable.

[2]Que hayamos simplificado no significa que esta explicación haya sido fácil, no te sientas idiota si te llevó más de una iteración entenderla.

En este curso (y en general en cualquier ambiente de programación) **ESTÁ TOTALMENTE PROHIBIDO** el uso de variables globales, a menos que las mismas sean de tipo `const`.

Se acepta definir de forma global información que diferentes funciones puedan leer, pero no se acepta que las funciones modifiquen el estado global de la aplicación.

Capítulo 8

Punteros

8.1. Introducción

Hasta el momento vimos que en el lenguaje C hay una jerarquía donde el `main()` dispara llamadas a funciones que a su vez llaman a funciones, y también vimos que las funciones reciben como parámetros expresiones del lenguaje, donde las mismas se evalúan y ese resultado se utiliza para inicializar a las variables que son parámetro de las funciones. La memoria no sobrevive a las funciones y las funciones pueden devolver un único valor, por lo que practicamente hay un flujo de información unidireccional desde el alto nivel hacia el bajo nivel.

Hay veces que esto no alcanza, hay veces que esto no es eficiente, hay veces en la que la memoria tiene que poder persistir a una función... y hay veces donde pasan otras cosas y hace falta un mecanismo diferente de acceso a la memoria que los que ya vimos.

Un puntero es una variable que puede almacenar una posición de memoria.

Si se entendió el párrafo anterior entonces podemos terminar el apunte acá.

¿Por qué es que el apunte sigue entonces?, porque partiendo de esa idea sencilla de almacenar una posición de memoria se abren un montón de posibilidades y complejidades que vamos a explorar el resto del curso.

```
1 int x = 5;
2 int *p;
3 p = &x;
4 printf("%d\n", *p);
```

El código anterior imprime `"5\n"`. La variable `p` es de tipo `int *`, puntero a entero, un tipo de variable que puede almacenar la dirección de memoria de una variable de tipo entero. Obviamente no guardamos posiciones de memoria porque somos acumuladores compulsivos sino que la idea de guardar una dirección es para poder manipular esa dirección después.

La instrucción `&i` devuelve la posición de memoria de la variable `x`. Dado que `i` es entera y la variable `p` almacena posiciones de variables enteras la asignación es correcta.

Ahora tenemos a `p` que almacenó la dirección de memoria de `x`. La instrucción `*p` le pide al compilador que vaya a buscar qué hay en la posición de memora que almacena `p`. Es decir, esa expresión evalúa al entero que está guardado en la posición referenciada por `p`. Dado que `p` referencia a `i` ese entero será el entero almacenado en `x`, un `5`. En la figura 8.1 se ve un esquema de la memoria suponiendo que la variable `x` vive en la dirección de memoria `0xA8`.

Fuera de broma este ejemplo introdujo el 100 % del comportamiento de los punteros. No hay nada más allá de esto.

¿Por qué seguimos entonces?, porque estos dos operadores y este tipo de variables son los bloques para construir un montón de cosas.

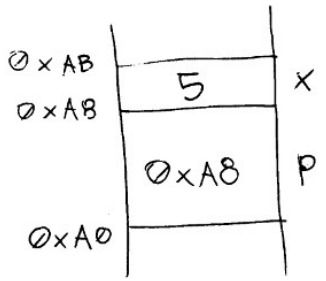

Figura 8.1: Representación de memoria de `int x; int *p = &x;`.

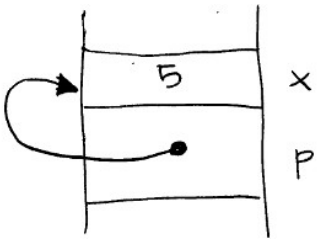

Figura 8.2: Esquematización con "flechas que apuntan" del ejemplo de la figura 8.1.

8.2. Nomenclatura

<T> *p;: p es un puntero a <T>, siendo <T> cualquier tipo. Esto quiere decir que p puede guardar direcciones de memoria de variables de tipo <T>.

&x: La dirección de x, operador de dirección, referencia la variable. Este operador sólo puede utilizarse sobre variables, porque son las únicas que tienen una dirección de memoria. Si $x \in$ <T> $\implies$ &x $\in$ <T> *. Por ejemplo, si x fuera de tipo **float**, entonces &x por ser la dirección de un **float** será de tipo **float** *.

p = &x: p apunta a x, obviamente el tipo de p tiene que ser tal que el tipo de &x sea el mismo.

***p:** El valor apuntado por p, operador de indirección, desreferencia la variable. Si $p \in$ <T>* $\implies$ *p $\in$ <T>. Por ejemplo, si p fuera un puntero a **float** entonces *p será un **float**.

Esta misma nomenclatura se utiliza en la representación pictórica de los punteros. Internamente la acción de apuntar es guardar una posición de memoria como se mostró en la figura 8.1, ahora bien, por simplicidad, para no tener que inventar direcciones de memoria ficticias podemos representar la acción de apuntar como en la figura 8.2. Ambas figuras representan lo mismo, pero es más sencillo representar el almacenamiento de una dirección de memoria como una flecha a lo referenciado. De esta representación viene el concepto de "apuntar".

8.3. Devolver valores mediante punteros

Antes de complejizar más el tema, presentemos un uso práctico de punteros relacionado. Como sabemos una función puede hacer **return** de un único valor.

¿Se puede en C devolver más de una cosa en una función? Utilizado **return** no. Ahora bien, podemos hacer que una función devuelva múltiples cosas y de forma explícita como nos impone el paradigma procedural.

Antes de ir a punteros presentemos un ejemplo sencillo de una función que devuelve un valor con **return** que llamaremos "devolución por el nombre":

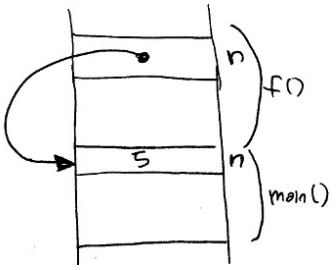

Figura 8.3: Esquema de la memoria del ejemplo al invocar a la función `f()`.

```
1  int f() {
2      return 5;
3  }
4
5  int main() {
6      int n;
7      n = f();
8      printf("%d\n", n);
9      return 0;
10 }
```

A estas alturas no necesita mayores exposiciones.

Imaginemos que ahora necesitamos que esta función devuelva más de una cosa y que la otra cosa es suficientemente importante como para merecerse el **`return`**. ¿Cómo podemos hacer para devolver nuestro `5` sin **`return`**?

El siguiente ejemplo es la reversión de este código sin **`return`**, utilizando punteros, que llamaremos "devolución por la interfaz":

```
1  void f(int *n) {
2      *n = 5;
3  }
4
5  int main() {
6      int n;
7      f(&n);
8      printf("%d\n", n);
9      return 0;
10 }
```

Puede observarse el esquema de la memoria en la figura 8.3.

Antes de decir otra cosa lo importante es ambos ejemplos implementan **la misma** función. `f()` en ambos casos es una función que tiene que devolver un valor entero. En un caso devuelve su valor entero por el nombre, en el otro devuelve su valor entero por la interfaz. No son dos funciones diferentes que hacen cosas distintas.

Dicho esto, entonces debería ser inmediato que el `main()` tiene que ser idéntico en ambos casos tanto en cómo va a declarar la variable receptora del resultado de invocar a `f()` como su manipulación posterior. Lo **único** que va a ser diferente es cómo realizar la llamada.

En el primer caso `n = f();`, en el segundo `f(&n)`. En un caso se guarda en `n` lo que devuelve `f()`, en el otro se le pasa a `f()` la posición de memoria en la que queremos que `f()` guarde el resultado, que no es otra cosa que la dirección de memoria de `n`. En la devolución por la interfaz delegamos la asignación a la función.

El parámetro `n` de la versión con punteros es un parámetro *de salida*, es decir, la función no va a leer de ese parámetro ningún dato relevante para su operatoria, es simplemente la referencia de dónde guardar el resultado de su cómputo.

Es importante notar que si una función quiere devolver un **`int`** entonces la firma de la función necesita recibir un **`int *`** porque lo que necesita es que le pasen la posición de memoria de un entero en el que va a guardar. ¿Por qué?, porque si no no podría resoverse el problema. Mirar el siguiente ejemplo:

```
1  void f(int n) {
2      n = 5;
3  }
4
5  int main() {
6      int n;
7      f(n);
8      printf("%d\n", n);
9      return 0;
10 }
```

¿Estamos de acuerdo en que va a imprimir basura? Si no entendés por qué, volvé a leer el capítulo anterior sobre alcance de variables.

Retomando, para la devolución por la interfaz necesito un nivel adicional de punteros.

8.4. Punteros al inicio de un arreglo

Supongamos que tenemos un arreglo

```
1  int valores[] = {10, 20, 30, 40};
```

cada uno de sus elementos es un **`int`** por lo que podríamos querer apuntar un puntero a alguno de sus elementos.

Apuntemos un puntero a su primer elemento:

```
1  int *p = valores;
```

¿No falta un `&` por ahí? No.

El nombre de un arreglo es un puntero a su primer elemento. En código: `valores == &` ↪ `valores[0]`

¿Esto de que el nombre de un arreglo es un puntero a su primer elemento vale para cuando lo asignamos a un puntero? No, vale siempre. En cualquier expresión donde aparezca el nombre de un arreglo **siempre** eso va a evaluar como un puntero al primer elemento.

Recordemos el capítulo "El problema del **`sizeof`** de los arreglos" (y si no te acordás, releelo ahora, es relevante) ahí habíamos presentado el siguiente código:

```
1  #include <stdio.h>
2
3  void f(int valores[]) {
4      printf("%p\n", valores);
5      printf("%zd\n", sizeof(valores));
6  }
7
8  int main(void) {
9      int valores[] = {1, 2, 3, 4, 5};
10
```

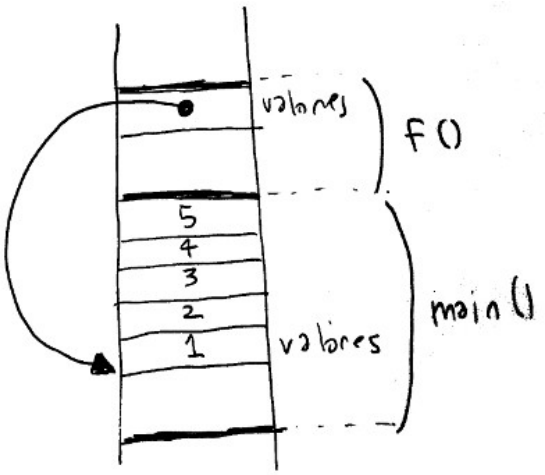

Figura 8.4: Esquema de la pila de memoria en la llamada a `f()`.

```
11      printf("%p\n", valores);
12      printf("%zd\n", sizeof(valores));
13
14      f(valores);
15
16      return 0;
17  }
```

y habíamos hablado de la extrañeza de que adentro de la función el **`sizeof`** valía un inexplicable 8.

Miremos en detalle la llamada a la función `f(valores)`... estamos invocando la función con el nombre del arreglo, es decir, estamos pasando un puntero al primer elemento (figura 8.4). La realidad es que la variable `valores` de la función `f()` no es de "tipo arreglo" no es otra cosa que un **`int`** `*`, en prototipos de funciones **`int`** `v[]` es lo mismo que **`int`** `*v`.

Notar también el detalle de `printf("%p\n", valores);`, el modificador `"%p"` que servía para imprimir posiciones de memoria (la p es de puntero) está recibiendo como parámetro `valores`, es decir, la posición de memoria del primer elemento.

Es mas, en C no hay manera ni de evaluar ni de asignar un arreglo a ninguna cosa, cada vez que hacemos una operación con arreglos estamos operando con un puntero al primer elemento.

8.5. Aritmética de punteros

Supongamos el siguiente ejemplo:

```
1  int valores[] = {10, 20, 30, 40};
2  int *p = valores;
```

al igual que antes, `p` apunta al primer elemento de `valores`. Por lo tanto `*p == 10`.

Ahora bien, la expresión `p + 1` está sumando el contenido de `p`, que es una dirección de memoria, con un número. Si `p` valiera por ejemplo 42, ¿cuánto valdrá `p + 1`? Bueno, valdrá 46. ¿Qué?, sí, 46, las operaciones de punteros se hacen en función del **`sizeof`** del tipo base apuntado. Como `p` es un **`int`** `*` se adiciona en unidades de **`sizeof`**`(`**`int`**`)`, por lo tanto de entero en entero, o de 4 en 4 bytes.

Entonces si `p` apunta al primer elemento de `valores`, `p + 1` apunta al segundo elemento de valores. `*(p + 1)== 20`.

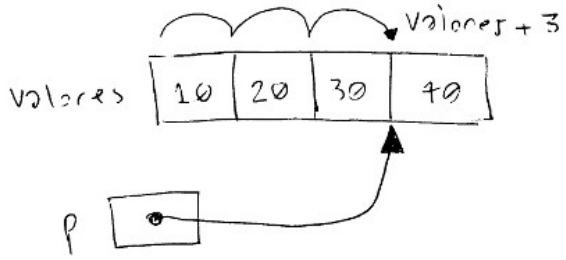

Figura 8.5: Esquema del puntero `int *p = valores + 3;`.

¿Se acuerdan del operador `[]` en C? No existe. Bueno, sí, existe, pero es lo que se denomina "azúcar sintáctica". En programación se le dice azúcar sintáctica a expresiones que existen sólo para hacernos más fácil escribir algo más complejo, pero en realidad el compilador reemplaza por lo que realmente quiere decir. Ya vimos un ejemplo de azúcar sintáctica en el capítulo anterior: Cuando declaramos una función `void f(int v[]);` para el compilador es transparente que estamos queriendo decir `void f(int *v);`, esa azúcar nos permite manipular vectores de forma totalmente inocente sin necesidad de conocer los detalles de punteros que hay detrás. Lo mismo vale para el operador `[]`.

La expresión `p[i]` es azúcar sintáctica para `*(p + i)`. Si el compilador ve la primera la reemplaza por la segunda. Es decir, el acceso a índices de vectores no es otra cosa que hacer operaciones de aritmética de punteros[1].

Como `p[i]` es lo mismo que `*(p + i)` y como la primera es mucho más fácil de entender (prueba: la entendiste sin saber nada de punteros) vamos a preferir la primera.

Entonces, volviendo a nuestro ejemplo, dado que `p` apunta a `valores`, `p[1]` es lo mismo que decir `valores[1]`, y en ambos casos el compilador está haciendo operaciones de punteros.

Entonces, aritmética de punteros: `p + i`, donde `p` es una expresión de tipo puntero e `i` es una expresión `size_t` incrementa `p` en `i` unidades de su tipo base.

También es una operación válida la resta de punteros `p - q` **siempre y cuando** `p` y `q` apunten al mismo bloque de memoria. El resultado es un `size_t` con la cantidad de unidades del tipo base que haya entre ambos. Ejemplo:

```
1 int valores[] = {10, 20, 30, 40};
2 int *p = valores + 3;   // p apunta a valores[3]
3
4 size_t n = p - valores; // n vale 3
```

(Ver figura 8.5.)

8.6. La memoria "data"

(Prometemos que es la última vez que decimos esto: Acá es donde vamos a terminar de explicar lo último que faltaba explicar del hola mundo.)

Comparemos estas dos declaraciones:

```
1 char a[] = "hola";
2 char *p = "hola";
```

En la primera estamos poniendo en nuestro stack un arreglo `char [5]` inicializado en `{'h', 'o', 'l', 'a', '\0'}`. En la segunda lo que tenemos en el stack es un `char *` que almacena una dirección de memoria ¿Adónde?

[1] Como en realidad ya habíamos presentado en "El problema del `sizeof` de los arreglos" que se supone que acabás de releer.

Cuando se inicia nuestro programa el mismo carga en memoria un bloque que se llama "data", que es de solo lectura, y que contiene información estática que va a ser utilizada en nuestro programa. Los literales que escribimos entre comillas viven en ese área de la memoria.

¿Entonces, si `a` ocupa 5 bytes y `p` ocupa 8 bytes es más eficiente usar `a` que usar `p`? No, todo lo contrario. Es cierto que `a` ocupa 5 bytes en el stack, pero si el marco de las funciones se crea cuando las invoco, ¿cómo se inicializa la variable `a` cada vez que ejecuto mi función? Fácil, la cadena `"hola"` se encuentra en el data. Cada vez que se invoca la función el mi programa tiene que copiar esa memoria desde ahí al stack para inicializar mi variable.

En el caso de `p` también hay una inicialización: Asignar el puntero que corresponda del data en `p`.

¿Entonces está mal la declaración de `a`? No, simplemente son cosas diferentes. Ya dijimos que data era una zona de memoria de sólo lectura[2], si nosotros necesitaríamos modificar el contenido de la cadena, necesitamos tenerla en el stack.

Es redundante decirlo, pero en:

```
1 int valores[] = {10, 20, 30, 40};
```

también los 16 bytes de inicialización de `valores` viven en data y mi programa tiene que copiar eso al stack al iniciar mi función. Si esos valores fueran constantes, evaluá si no te conviene declarar el vector como una variable global **`const`** para garantizar que no se ponga en memoria más de una vez.

8.7. Punteros a `void`

Si las variables de tipo puntero almacenan direcciones de memoria, ¿en qué se diferencian las direcciones de memoria de un **`int`** de las de un **`float`**? En nada.

Si simplemente necesitamos almacenar una dirección de memoria podemos hacerlo sin especificar el tipo:

```
1 int i;
2 void *p = &i;
```

`p` será un "puntero a **`void`**" que está apuntando a `i`.

Ahora bien, en un **`void *`** tengo limitadas dos operaciones básicas de punteros: No puedo hacer `*p` porque el compilador no puede resolver de qué tipo es esa memoria, y por lo tanto no puede traducir eso en operaciones del procesador. Y tampoco puedo hacer `p + i`, porque no sabe cuál es el **`sizeof`** de lo asignado.

Como ya se dijo: Sirven únicamente para almacenar una dirección. No sirven para manipular la memoria en esa posición.

8.7.1. `memcpy()`, `memmove()` y `memcmp()`

Como ejemplo de la utilidad de **`void *`** están estas tres funciones de `<string.h>`.

Imaginemos el siguiente problema: Queremos copiar los elementos de un vector de enteros en otro vector. Sabemos hacer eso. Ahora bien, queremos copiar los elementos de un vector de flotantes en otro vector. Sabemos hacerlo también. Y también sabemos implementar una función que copie en otro vectores de un tipo cualquiera.

Ahora bien, en C cada función es una función diferente. Pongamos de ejemplo:

```
1 void copiar_vint(int destino[], const int origen[], size_t n) {
2     for(size_t i = 0; i < n; i++)
```

[2] Si no lo fuera, ¿quién me garantizaría que después de ejecutar mi programa un rato en ese lugar siguiera diciendo `hola`?

```
3         destino[i] = origen[i];
4 }
```

Esta función no es una función genérica, recibe dos punteros a entero y al realizar la copia hace `destino[i] = origen[i]`, una operación de indirección, que implemente la copia de una variable entera en otra variable entera según las reglas de los enteros. ¿Cuál sería el resultado final de la operación? En el final el vector `destino` será una copia idéntica del vector `origen`, byte a byte.

Y si al final de cuentas los vectores se habrán copiado y serán idénticos en memoria, ¿era relevante saber que la memoria contenía enteros o tengo algún mecanismo para copiar los bytes que la componen y ya?

Esta es una aplicación para **`void *`**, en vez de plantear una función que copie enteros, flotantes u otro tipo puedo pensar a la memoria como bytes. Pensemos ahora el siguiente ejemplo:

```
1 void copiar(void *destino, const void *origen, size_t n) {
2     char *d = destino;
3     char *o = origen;
4     for(size_t i = 0; i < n; i++)
5         d[i] = o[i];
6 }
```

Esta función copia bytes, es cierto, lo hace con instrucciones de **`char`**, pero dado que la asignación dentro del mismo tipo no realiza ninguna conversión, preserva el contenido de los bytes representen lo que sea que representen. **Es importante** notar que en este caso `n` ↪ no representa la cantidad de elementos de los vectores sino la cantidad de bytes. Por ejemplo, si quisiéramos copiar un vector de 5 enteros deberíamos invocar a la función con `5 * sizeof(int)`.

En este caso de la copia está implícito que **no** me interesa entender el contenido del vector. Es decir, el acceso a la memoria de los punteros es sólo a fines de mirar los bytes. La función no puede adivinar de qué tipo es la memoria. Digamos, esta técnica nos resuelve el problema de copiar memoria pero no nos serviría para, por ejemplo, sumar los elementos de un vector de tipo desconocido. Eso requiere aplicar reglas de operaciones de procesador que son específicas de los tipos.

La biblioteca estándar en el encabezado `<string.h>` provee las funciones:

`void *memcpy(void *destino, const void *origen, size_t n`: Exactamente la función que acabamos de implementar. Eso sí, tiene un **`return`** `destino;` en la última línea, por eso el tipo de retorno.

`void *memmove(void *destino, const void *origen, size_t n`: Idéntica a la anterior... pero sabe detectar si hay un solapamiento entre ambos punteros. Por ejemplo, si invocáramos `memmove(v + 1, v, 10 * sizeof(int));` y aplicáramos el algoritmo que ya implementamos no haríamos otra cosa que copiar `v[0]` 10 veces. Queda como ejercicio entender esto y pensar la solución. Por lo general vamos a preferir utilizar esta función sobre la anterior, como dijimos, hacen lo mismo.

`int memcmp(const void *v1, const void *v2, size_t n);`: Compara los bytes de ambos vectores, devuelve un entero menor que, igual a o mayor que cero dependiendo de si el primer byte que difiere es respectivamente menor, igual o mayor en `v1` que en `v2`. Es decir, devuelve `0` sólo si recorre los `n` bytes y son todos iguales.

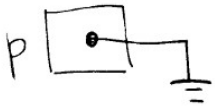

Figura 8.6: Representación gráfica de un puntero a NULL.

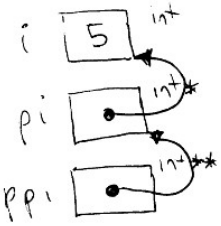

Figura 8.7: Esquema del puntero a puntero del ejemplo.

8.8. El puntero NULL

En muchas aplicaciones es necesario poder tener un centinela que me diga que una variable de tipo puntero no apunta a ningún lugar. Para eso el estándar provee la etiqueta NULL. Ejemplo:

```
1 int *p = NULL;
```

Sé que el puntero p no apunta a ninguna posición de memoria válida. Notar que no es lo mismo que no inicializar p, si no inicializo p dado que p se inicializa en basura tampoco apuntará a una posición de memoria válida... pero no tengo manera de validarlo.

Vamos a utilizar a NULL cuando tengamos que avisarle a una función que le pasamos un puntero no válido o cuando una función nos quiera devolver un puntero no válido y que nosotros sepamos que no es válido. Por ejemplo, la función fgets() devuelve un **char** *, y ya vimos que al alcanzar la marca de fin de archivo la misma devuelve NULL.

Gráficamente vamos a representar los punteros a NULL con el símbolo de una puesta a tierra eléctrica (figura 8.6).

8.9. Punteros a punteros

Como se dijo, una variable de tipo puntero a cosa es capaz de almacenar posiciones de memoria de *cosas*. Pero a su vez ella es una variable, por lo que tiene una dirección de memoria.

```
1 int i, *pi, **ppi;
2 pi = &i;
3 ppi = &pi;
4
5 **ppi = 5;  // Asigno en i
```

(Ver figura 8.7.)

Siguiendo lo ya dicho, si pi es una variable de tipo **int** * entonces &pi será una variable de tipo **int** **.

El tipo **int** ** es un *"puntero a puntero a entero"*, es decir, una variable que guarda direcciones de memoria de tipo puntero a entero. Ahora bien, usualmente, de forma coloquial vamos a referirnos a él como *"doble puntero a entero"*.

Esta última nomenclatura no significa nada y se adopta pura y exclusivamente por un tema de comodidad... ¿cómodidad dónde?, comodidad cuando tengamos punteros de orden mayor. Por ejemplo, una variable de tipo **int** **** formalmente será un *"puntero a puntero a puntero*

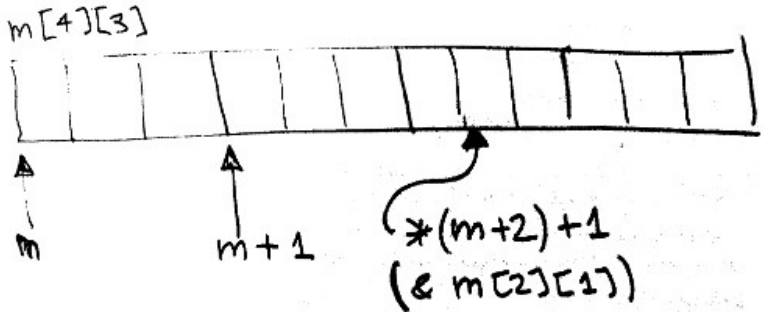

Figura 8.8: Esquema del las operaciones de punteros en una matriz `m[4][3]`.

a puntero a entero", ¿no sería mucho más práctico llamarla *"cuádruple puntero a entero"*? Así lo haremos. Y sí: Van a aparecer punteros de órdenes altos más adelante.

8.10. Matrices

En la sección 6.4 vimos las matrices de C, las cuales sabemos que se definen como `tipo matriz[F][C]` y se accede a sus elementos utilizando dos subíndices: `matriz[i][j]`. Ahora bien, a la vista de lo visto en este capítulo el operador corchetes no es otra cosa que azúcar sintáctica para un acceso a punteros: `*(*(matriz + i)+ j)`.

Esta última expresión es confusa, si el operador `*` aparece dos veces aplicado en una expresión, ¿entonces `matriz` es de tipo `int **`? La respuesta es no, y la explicación es confusa también.

Más allá de qué hace el lenguaje para justificar esos dos asteriscos, la explicación de cómo hace el compilador para acceder a un elemento y qué aritmética realiza la vimos detallada en la sección antedicha. Si no te acordás, repasala.

Recordemos que `matriz[i]` se traducía como *"la iésima fila de la matriz"* entonces en la expresión `*(matriz + i)` matriz tiene que ser un puntero a "filas de matriz"... y efectivamente eso es. De hecho el nombre de un arreglo es un puntero a su primer elemento `matriz == &matriz[0]` su tipo es puntero a filas.

```
1 int matriz[F][C];
2 int (*p)[C] = matriz;
```

El tipo de `p` es *"puntero a arreglos de C enteros"*, lo cual tiene sentido si habíamos presentado a las matrices como arreglos de filas, cada uno de sus elementos será un arreglo de tantos elementos como columnas.

Ya que estamos, sabíamos que si tenemos una función `void foo(int v[]);` eso es azúcar sintáctica para `void foo(int *v)`. Análogamente, una función `void foo(int m[F][C]);` es azúcar sintáctica para `void foo(int (*m)[C]);`. Eso explica lo ya visto en la sección 6.4.1.

Entonces, si cuando pasamos matrices a funciones los tipos serán como el tipo de `p` en nuestro ejemplo, sabemos que `p` se puede utilizar para acceder a los elementos de una matriz con doble índice.

Ahora sí entonces si `p` es puntero a fila, `*(p + i)` será moverse `i` unidades de `sizeof(int [C])`[3] en memoria y desreferenciar (ver figura 8.8). El resultado de eso será un arreglo de `C` enteros, o mejor dicho, un puntero al primer elemento de ese arreglo. Es decir, el operador asterisco en este caso lo único que hace es cambiar el tipo del puntero, no hace ningún tipo de acceso a memoria ni desreferencia nada. Las expresiones `p + i` y `*(p + i)` evalúan a la misma

[3]Que es lo mismo que decir `C * sizeof(int)`.

dirección de memoria, sólo que la primera es de tipo `int (*)[C]` mientras que la segunda de tipo *int**, que es el tipo con el que apuntamos arreglos.

A partir de ahí el segundo índice `j` es consistente con un puntero a entero.

Esto que se explicó hasta aquí es la operatoria que hace internamente el lenguaje para permitir el uso del doble índice tratándose de memoria unidimensional. Todo, todo, todo es azúcar sintáctica para no operar como realmente es que es:

```
1 int matriz[F][C];
2 int *p = &matriz[0][0];
3 p[i * C + j];   // Accedo a matriz[i][j]
```

Obviamente vamos a abrazar la azúcar sintáctica y usar los dobles corchetes.

8.11. Punteros a funciones

Si bien se incluyen dentro del mismo capítulo por un tema de orden, los punteros a funciones poco tienen que ver con los punteros. Los punteros que vimos hasta ahora pueden almacenar direcciones de memoria de datos y funcionalmente sirven para acceder a esa memoria y modificarla. Los punteros a funciones también almacenarán direcciones, pero en este caso no de datos sino de código y funcionalmente servirán para hacer llamadas a ese código.

Cuando nuestro código se compila y enlaza, el mismo forma una secuencia de instrucciones de código máquina, donde el procesador ejecuta la instrucción que le dice el *program counter*[4] (PC). Al realizar una llamada a una función se reemplaza el PC por la posición en código donde se encuentra esa subrutina. Esto dependerá de la arquitectura del procesador, pero no necesariamente la memoria de datos y la memoria de programa forman parte del mismo bloque de memoria. Los punteros a funciones almacenan la posición de funciones.

Más allá de la introducción, los punteros a funciones son sencillos de utilizar (aunque bastante feos de declarar):

```
1 int (*conversor)(int);      // conversor es un puntero a funciones
   ↪ de firma: int f(int);
2 conversor = toupper;        // int toupper(int) @ ctype.h:
   ↪ Convierte un carácter a mayúsculas.
3
4 putchar(conversor('a'));    // Imprime 'A'.
```

Antes de entrar en sintaxis, funcionalmente el código anterior define un puntero el cual se hace apuntar a la función `toupper()`, luego, invocar al puntero como si fuera una función es equivalente a llamar a la función `toupper()`. Como `conversor` es un puntero así como en el ejemplo apunta a `toupper()` podemos hacer que apunte a cualquier otra función que tenga la firma `int f(int);`, es decir, podemos usar ese puntero para apuntar a funciones diferentes según el contexto, utilizando el mismo código. **Atentos a la sintaxis:** Cuando asignamos el puntero decimos `conversor = toupper;` notar que no hay paréntesis, no estamos llamando a la función `toupper()`, sólo estamos escribiendo su nombre, eso constituye un puntero a ella.

Como invocar a un puntero a función implica llamar a una función, el compilador tiene que saber los parámetros de la función a ser llamada. Es por eso que los punteros a función pueden apuntar a funciones con determinados parámetros. Es decir, el tipo del puntero a función está dado por los parámetros que toma y recibe la función a ser apuntada. En el ejemplo estamos declarando `int (*conversor)(int);`, notar que esta sintaxis es similar al prototipo de una función, sólo que se encerró entre paréntesis y con un asterisco el nombre de la función. Si `int conversor(int);` declara una función `conversor` que toma un entero y

[4] Ver capítulo 7.2.

devuelve un entero, `int (*conversor)(int);` declara un puntero a función `conversor` que apunta a funciones que toman un entero y devuelven un entero.

¿Para qué sirven los punteros a funciones? Su uso principal es desacoplar problemas. Por ejemplo, quisiera convertir a mayúsculas una cadena de caracteres. Para resolver ese problema tendría que recorrer la cadena de caracteres y luego para cada carácter debería convertirlo a mayúsculas. Tal vez yo sé cómo convertir a mayúsculas pero no sé cómo recorrer una cadena[5] o viceversa. El código del problema es algo así como:

```
void convertir_a_mayusculas(char cadena[]) {
    for(size_t i = 0; cadena[i] != '\0'; i++)
        cadena[i] = toupper(cadena[i]);
}
```

Bien. Ahora, ¿cómo hacemos si queremos convertir a minúsculas? Podemos copiar y pegar la función completa y cambiar la llamada a `toupper()` por `tolower()`. ¿Y si quisiéramos hacer otra conversión diferente? A estas alturas del curso sabemos que repetir código nunca suele ser una solución correcta. Como decíamos antes, el problema de recorrer y aplicar algo en todos los elementos de una cadena es un problema diferente del qué aplicar. Si pudiéramos separar los problemas evitaríamos duplicar:

```
void convertir(char cadena[], int (*conversor)(int)) {
    for(size_t i = 0; cadena[i] != '\0'; i++)
        cadena[i] = conversor(cadena[i]);
}

...
convertir(cadena, toupper); // Convierte a mayúsculas
convertir(cadena, tolower); // Convierte a minúsculas
```

Incluso, imaginemos que queremos reemplazar todas las vocales por equis. Podemos hacer:

```
int censurar_vocales(int c) {
    if(c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u')
        return 'x';
    return c;
}

...
convertir(cadena, censurar_vocales);
```

Como dijimos: Desacoplamos el problema de recorrer la cadena del problema de cómo la transformamos. La función `convertir()` puede recibir la función que queramos que haga la conversión que nos interese.

8.11.1. `qsort()`

Si el ejemplo que mencionamos te pareció trillado cambiémoslo por un ejemplo más concreto. ¿Sabés ordenar los elementos de un vector de forma eficiente? Probablemente no. Ahora bien, si tuvieras dos elementos, ¿podrías decir cuál debería ir primero en un vector ordenado? Dicho de otra manera, el problema de ordenar no depende del criterio de ordenamiento, ¿no?

Por ejemplo, tenemos un vector de flotantes. Si quisiéramos ordenarlo de forma descendente para cualquier par de elementos a, b, si $a > b$ entonces a debería ir antes que b.

[5] Es un ejemplo, tratá de seguirlo... a estas alturas del curso todos deberíamos saber recorrer cadenas.

En la sección 8.7.1 vimos a la función `memcmp()` que comparaba dos vectores y devolvía un entero para codificar cuál era "menor" al otro. Bueno, lo que devolvía la función esa es la **convención** de C siempre que queramos hacer funciones que devuelvan cosas: Se reciben dos elementos a y b, si $a < b$ se devuelve un número menor a cero, si $a = b$ se devuelve cero y si $a > b$ se devuelve un número positivo. Donde cuando decimos menor o mayor queremos decir literalmente se ordenan antes o después.

Volviendo a nuestro problema de los flotantes, podríamos definir:

```
1 int comparar_descendente(const float *a, const float *b) {
2     if(*a > *b)
3         return -1;
4     if(*a < *b)
5         return 1;
6     return 0;
7 }
```

Recordar que queríamos ordenar de forma descendente, entonces nuestra relación de orden es al revés que lo esperado.

Sin saber ordenar bien podemos definir un criterio para ordenar. ¿Y cómo ordenamos? En `<stdlib.h>` tenemos definida la función `void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));`. La función recibe el puntero al comienzo de un arreglo, la cantidad de elementos del arreglo, el tamaño de cada uno de los elementos del arreglo y finalmente el puntero a la función que le dice cómo ordenar. Mientras `qsort()` ordene va a invocar a la función que le hayamos pasado pasándole el puntero a dos elementos y es la función la que le va a indicar cuál está antes de cuál.

Para ordenar un vector de flotantes de forma descendente, por ejemplo:

```
1 float v[100] = {...}
2
3 qsort(v, 100, sizeof(float), comparar_descendente);
```

Ahora bien, la firma de nuestra función no es idéntica a la de la función que espera recibir `qsort()` dado que espera recibir una con punteros a `void` y nosotros tenemos con punteros a `float`. Si lo compilamos y lo ejecutamos vamos a ver que compila... ahora bien, con una advertencia de compilación fuerte que nos dice que estamos usando punteros de tipo incompatible. Podemos resolver eso creando una función *"wrapper"* (envoltorio) que acomode los parámetros:

```
1 int comparar_descendente_void(const void *a, const void *b) {
2     return comparar_descendente(a, b);
3 }
```

y luego llamar a `qsort` con esta función, que es una función boba que no hace otra cosa que llamar a la otra.

8.11.2. `typedef`

Como ya vimos el uso de punteros a funciones es relativamente sencillo, pero sí es confusa la forma de declaración de las variables. Imaginá cómo se complicaría querer definir un vector de funciones de comparación para pasarle a `qsort()` por ejemplo. Para mejorar esto podemos hacer uso de `typedef`:

```
1 typedef int (*comparacion_t)(const void *, const void *));
```

Teniendo esta declaración puedo reescribir el prototipo de `qsort()` como `void qsort(void *base, size_t nmemb, size_t size, comparacion_t compar);`, ¿mejora?

Quiero un arreglo de funciones de comparación:

```
1 comparacion_t arreglo[2] = {comparar_descendente_void, strcmp};
```

¿Mejora?

Quiero llamar a `qsort()` sin escribir un wrapper:

```
1 qsort(v, 100, sizeof(float), (comparacion_t)comparar_descendente);
```

¿Mejora? Y sí, podría haber casteado en el ejemplo anterior... ¿cómo? Sí, mejora.

Capítulo 9

Estructuras y tipos enumerativos

Componen este capítulo los temas de estructuras, tipos enumerativos y tablas de búsqueda. Ahora bien, sólo vamos a desarrollar el primero de estos, dado que para el segundo hay un apunte preexistente en la sección de Material de la página web de la materia.

9.1. Estructuras

Hasta ahora vimos que podemos declarar variables sueltas o que podemos armar bloques de memoria consecutivos de n elementos del mismo tipo. Ahora bien, hay problemas en los cuales una misma entidad necesita múltiples variables de diferente tipo (o que no tiene sentido pensar como una secuencia como en el caso de los vectores). Por ejemplo, si quisiera representar a una persona ella tendría un nombre, un apellido, una fecha de nacimiento, un DNI, etc. Y, tal vez, si tuviera que representar a una persona podria hacerlo con una determinada cantidad de variables, pero si tuviera que representar a muchas personas ese enfoque sería desordenado. Para eso el lenguaje provee las estructuras.

La construcción

```
1 struct persona {
2     char nombre[30];
3     int dni;
4     edad_t edad;
5 };
```

define un nuevo tipo llamado **struct** `persona` que contiene dentro tres "miembros" que se van a comportar como un arreglo de 30 caracteres, un entero y lo que sea que represente el tipo `edad_t`[1].

Teniendo este tipo podemos declarar y definir:

```
1 struct persona persona1 = {"Juan", 42123435, 20};
2
3 struct persona persona2 = {
4     .nombre = "Maria",
5     .dni = 92134245,
6     .edad = 19,
7 };
```

[1] Recordar la sección 3.7.

En el primer caso tenemos que enumerar las diferentes inicializaciones de los miembros en el mismo orden en el que se declararon en la definición del tipo, mientras que en el otro nos independizamos.

Una vez creada la variable podemos acceder a los miembros con el operador `.`:

```
printf("Nombre: %s, DNI: %d, Edad: %d\n", persona1.nombre,
    ↪ persona1.dni, persona1.edad);
```

Similar a los arreglos, el lenguaje C no provee herramientas para manipular una estructura. Pero cada uno de los miembros se comporta como una variable de un tipo conocido para el cual C sí provee estas herramientas.

9.1.1. `typedef`

Notar que el tipo de las estructuras se compone de dos palabras: **`struct`** más el nombre de la estructura. Muchas veces por comodidad se prefiere utilizar un **`typedef`** para englobar al tipo:

```
typedef struct persona persona_t;
```

Incluso podría ya declararse la estructura con **`typedef`**:

```
typedef struct {
    char nombre[30];
    int dni;
    edad_t edad;
} persona_t;
```

Notar que en este caso el nombre del tipo va después de la llave que concluye la estructura, a diferencia de la definición vista antes.

9.1.2. Asignación de estructuras

En el lenguaje C la asignación de una estructura en otra es azúcar sintáctica para una llamada a `memcpy()`:

```
persona_t a = {"Juan", 42123435, 20};

persona_t b = a;    // Equivalente a memcpy(&b, &a, sizeof(
    ↪ persona_t));
```

Notar que, a contramano del criterio minimalista de las operaciones de C, una asignación de estructuras puede ser tremendamente ineficiente dado que el **`sizeof`** de una estructura puede ser arbitrariamente grande.

Esto toma relevancia sobretodo cuando implementamos funciones:

```
void imprimir_persona(persona_t persona) {
    printf("Nombre: %s, DNI: %d, Edad: %d\n", persona.nombre,
        ↪ persona.dni, persona.edad);
}
```

En esta función cada vez que la misma sea invocada se copiará en la pila cada uno de los bytes de la variable que viva en la función invocante.

Es por esto que **prácticamente siempre** pasaremos las estructuras a las funciones a través de punteros:

```
1 void imprimir_persona(const persona_t *persona) {
2     printf("Nombre: %s, DNI: %d, Edad: %d\n", (*persona).nombre,
      ↪ (*persona).dni, (*persona).edad);
3 }
```

Notar como pasamos de copiar en el stack algo que incluye un vector de 30 caracteres a simplemente recibir una dirección de memoria, lo que es justamente el comportamiento que C tiene para los arreglos.

9.1.3. Punteros a estructuras

Ya vimos que para acceder a un miembro de una estructura tenemos el operador `.`, y en el ejemplo anterior vimos que la sintaxis se empasta cuando tenemos punteros a estructuras:

```
1  persona_t a;
2  persona_t *b = &a;
3
4  a.dni = 12678145;
5  (*b).dni = 12678145;
6  \begin{lstlisting}
7  (Los paréntesis son necesarios por precedencia de operadores.)
8
9  Bueno, para simplificar la sintaxis el lenguaje nos provee otra az
   ↪ úcar sintáctica como la que nos dió para acceder a elementos
   ↪  de punteros:
10 \begin{lstlisting}
11 b->dni = 12678145;
```

El operador flecha[2] `a->x` no es otra cosa que azúcar sintáctica para `(*a).x`.

Para distinguir rápido cuándo se usa punto y cuánto se usa flecha simplemente hay que recordar si tenemos una estructura o un puntero a estructura. Si tenemos estructura va punto, si tenemos puntero a estructura va flecha.

9.1.4. Tamaño de las estructuras (alineación)

Supongamos el siguiente ejemplo:

```
1  typedef struct {
2      signed char piso;
3      int numero;
4  } aula_t;
5
6  int main(void) {
7      aula_t aula = {2, 202};
8
9      printf("%zd\n", sizeof(aula.piso));      // Imprime 1
10     printf("%zd\n", sizeof(aula.numero));    // Imprime 4
11     printf("%zd\n", sizeof(aula));           // Imprime 8 !!!
12
13     return 0;
14 }
```

[2]Lo llamamos así pero es un guión seguido de un signo de mayor que.

La memoria que contiene a la estructura tiene que tener capacidad al menos para cada uno de sus miembros, pero no hay garantías de que tenga exactamente ese tamaño[3].

La explicación de este comportamiento tiene que ver con detalles de implementación de hardware. Sin ahondar de forma superficial podemos decir que los diseñadores de procesadores alguna vez dijeron, si los enteros miden 4 bytes y la gente va a poner muchos enteros en memoria... ¿por qué no forzamos a que los acomoden en posiciones múltiplas de 4 de memoria? Y podremos preguntarnos qué se gana con eso, bueno, si los enteros están en posiciones múltiplas de 4, los últimos dos bits de esa posición van a ser siempre cero. Y si son siempre cero podemos ahorrarnos de routear dos vías en todos los buses de datos. Eso es una reducción importantísima en la complejidad del procesador, ¿por qué no hacerlo?

Ahora bien, la contrapartida de forzar a que la memoria esté en posiciones particulares (esto se llama "alineación") es que el compilador se ve forzado a dejar espacios (que se llaman "padding") entre las variables para que todas estén alineadas según su tipo.

En nuestro ejemplo, podríamos suponer que entre el `piso` y el `aula` hay 3 bytes de padding para que si la estructura comienza en una posición múltiplo de 4, entonces la variable entera también lo haga.[4]

En nuestro curso vamos a asumir que **SIEMPRE** hay paddings en las estructuras, que es la postura más independiente de la plataforma que podemos tomar.

9.1.5. Asignación (otra vez) y comparación

Volviendo a las operaciones, ya sabemos que la asignación de estructuras es azúcar sintáctica para `memcpy()`, es decir cuando decimos `b = a` (siendo ambas estructuras) se copia toda la memoria de `a` en `b`, y no hay nada de malo en eso. Una será una copia idéntica de la otra y serán iguales. Pero tenemos que tomar en cuenta que esta operación copia todo, inclusive el padding que contiene basura[5].

Otra vez: No hay **nada** de malo con esto. Sólo estamos describiendo el comportamiento. (Y tal vez sí habría algo malo si, por ejemplo, sería más eficiente copiar la cadena con `strcpy()` para economizar movimiento de memoria.)

Ahora bien, ¿qué pasa al hacer?:

```
1 if(a == b) {
2     ...
3 }
```

Análogamente a la asignación, la comparación es en realidad una llamada a `memcmp()`. Y ahí sí **está mal**. Si comparamos la memoria de `a` y `b` byte a byte estaremos comparando no sólo el nombre, el DNI y la edad, si no además toda la basura que viene en la memoria, tanto en el padding como en lo que haya después del `'\0'` de las cadenas.

El operador de comparación podría decirme que son diferentes dos estructuras que tienen exactamente lo mismo asignado en sus miembros. La **única** manera de comparar estructuras es haciéndolo miembro a miembro:

```
1 if(a.dni == b.dni && a.edad == b.edad && strcmp(a.nombre, b.nombre
    ↪ ) == 0) {
2     ...
3 }
```

[3] No confundir este 8 con el 8 que habíamos visto en la sección 6.3, el 8 en este ejemplo es casualidad.

[4] Y podés preguntarte por qué no acomodar la estructura en memoria de tal manera que la variable entera quede en una posición válida y usar 5 bytes. Bueno, la respuesta tiene que ver con arreglos. Un arreglo es una sucesión de variables consecutivas en memoria, no habría manera de que dos estructuras de 5 bytes que contienen un entero adentro se puedan poner consecutivas en memoria y que el entero quede alineado.

[5] Y dicho sea de paso, en nuestros ejemplos teníamos `char nombre[30] = "Juan";`, también hay 25 bytes de basura ahí después del `'\0'`.

9.2. Tipos enumerativos

Ver el apunte en la web de la materia.

9.3. Tablas de búsqueda

Ver el apunte en la web de la materia.

Capítulo 10

Manejo de bits

Omitiremos este capítulo dado que ya hay un apunte sobre el tema en la página web de la materia.

Capítulo 11

Memoria dinámica

11.1. Introducción

Hasta el momento hablamos de diferentes lugares de la memoria: El stack, la memoria data, el lugar donde viven las variables globales. Toda esa memoria forma parte del mismo bloque de memoria: La memoria que el sistema operativo le asigna de forma estática a cada proceso. Es decir, cada vez que se arranca una instancia de un programa en una computadora el sistema operativo le reserva una equis cantidad de memoria para que ahí adentro el programa haga lo que quiera. Esta cantidad dependerá de la configuración, pero es un tamaño de no más de un par de megabytes. Ahora bien, si bien esta suma de memoria puede sobrar para algunas aplicaciones, para muchas será insuficiente.

Por fuera de esta limitación el modelo de memoria de stack que estamos viendo tiene una limitación importante en el ámbito de vida de la memoria. Las funciones de más alto nivel pueden compartirle su memoria a las de más bajo nivel, a través de punteros, pero lo contrario no funciona. Dado que la memoria donde viven las funciones es volátil, si una función de bajo nivel devolviera un puntero ese puntero pasaría a ser inválido en el mismo momento en el que la función termina, dado que el stack se desocupa. Esto plantea un tema de diseño, donde si se requiere que una función de bajo nivel manipule memoria es la función invocante la que tiene que prever reservar esa memoria previamente, con lo cual se invirten las responsabilidades, porque para conocer el tamaño de esa memoria hay que conocer el problema a resolver lo cual viola la abstracción de delegar en funciones.

Estos dos párrafos plantean tres problemas donde el esquema de memoria visto es insuficiente:

- Cuando el problema es grande.
- Cuando no puedo prever el tamaño del problema (y no puedo apostar a un número grande por el ítem anterior).
- Cuando requiero que una función devuelva memoria que le sobreviva.

11.2. El *heap*

Como se dijo, cuando el sistema operativo crea un proceso le asigna una porción de memoria de tamaño fijo que es lo que denominamos el stack. Por diseño el stack tiene que tener un tamaño pequeño dado que cada proceso utiliza esa memoria y puede haber indefinidos procesos. Ahora bien, hablamos de que el stack de cada aplicación suele ocupar un par de megabytes, pero una computadora tiene memoria que se contabiliza en gigabytes. La memoria que no

constituye el stack se denomina *"heap"*[1], y podemos utilizarla gestionándola con el sistema operativo.

A diferencia del stack que es fijo, equitativo para todos los procesos y está garantizado[2] en el heap podemos pedir la cantidad de memoria que queramos, cada proceso gestiona la propia (si la necesita) y no hay ninguna garantía de que esa memoria esté disponible.

El protocolo para la administración de la memoria es sencillo: Se le pide al sistema operativo una determinada cantidad de bytes de memoria. Si el sistema operativo los tiene, reserva esa memoria para nuestro proceso y nos entrega un puntero al primer byte de ese bloque de memoria. Cuando nosotros no necesitemos más la memoria, tenemos que devolvérsela al sistema operativo. Una vez devuelta la memoria la misma ya no nos pertenece.

En este curso **insistiremos particularmente** en dos cosas: Todo pedido de memoria puede fallar y toda la memoria que se pida debe ser devuelta.

Cabe preguntarse qué pasa con la memoria de heap que pidió un proceso cuando el mismo termina. En un sistema operativo moderno el mismo debería liberar por sí solo todos los recursos asociados a ese proceso. Ahora bien, para nuestra filosofía, no nos importa qué haga el sistema operativo después. Si un recurso lo reservamos explícitamente, ese mismo recurso lo vamos a liberar explícitamente. Además, estamos en un curso para alumnos de Ingeniería Electrónica, las aplicaciones que hacemos los electrónicos generalmente nunca terminan mientras el dispositivo esté energizado.

11.3. `malloc()` y `free()`

Como ya se dijo los pedidos de la memoria se hacen por una determinada cantidad de bytes. Ahora bien nosotros pedimos memoria de forma utilitaria porque necesitamos almacenar valores de determinado tipo. Tiene sentido entonces que ese pedido lo pensemos como unidades de determinado tipo, y además, la referenciemos con punteros consistentes.

El siguiente código:

```
1 #include <stdio.h>
2 ...
3 int *v = malloc(20 * sizeof(int));
4 ...
5 free(v);
```

pide la memoria suficiente como para un bloque de `20` enteros en memoria, la función `malloc()` hace ese pedido. Como se trata de enteros los vamos a referenciar con el puntero `v` que es a enteros. Luego del pedido de memoria podremos manipular este bloque como si fuera un arreglo de 20 elementos enteros... que de hecho lo es, la única diferencia es que vive en el heap en vez del stack, cosa que es indistinguible para nosotros. Finalmente cuando no necesitemos más esa memoria tendremos que liberar los recursos con `free()`.

Como regla mnemotécnica, un pedido de memoria siempre se ve `tipo *p = malloc(n * sizeof(tipo));`. Tiene sentido, si quiero pedir memoria para elementos de tamaño `tipo` y la función me va a devolver un puntero a uno de ellos necesito apuntarlo con un `tipo *`. Dependiendo del problema `n` tal vez valga 1 y se omita. Y dependiendo del problema, si justo estuviera pidiendo memoria para `char` podríamos omitir el `sizeof(tipo)` dado que es el único caso en el que está garantizado que valga 1. Por lo general, para hacer más uniformes los

[1] Y no tenemos una buena traducción al castellano, así que utilizaremos la palabra en inglés. Literalmente heap es montón, o montículo.

[2] Si el sistema operativo se quedara sin espacio para el stack, directamente no podría lanzar el proceso, así que si hay proceso es porque había espacio para su stack.

pedidos eligiremos ponerlo[3].

En la sección anterior dijimos que en este curso insistiríamos particularmente con dos cuestiones. La primera era liberar la memoria, cosa que hicimos, la segunda era que los pedidos de memoria podían fallar, cosa que no hicimos. La función `malloc()` devuelve un puntero válido cuando hay memoria disponible y `NULL` cuando el sistema operativo no puede satisfacer nuestro pedido. Qué hacer ante una falla de memoria dependerá del problema, pero validarlo y tomar una decisión es imprescindible.

En el caso anterior deberíamos tener un código de manejo de error:

```
1  int *v = malloc(20 * sizeof(int));
2  if(v == NULL) {
3      // Error
4  }
```

De más está decir que si falla el pedido de memoria no hay nada que liberar.

Supongamos una función que recibe una cadena de caracteres y devuelve una cadena igual pero en mayúsculas:

```
1  char *cadena_a_mayusculas(const char *s) {
2      char *nueva = malloc(strlen(s) + 1);
3      if(nueva == NULL)
4          return NULL;
5
6      for(size_t i = 0; (nueva[i] = toupper(vieja[i])); i++);
7
8      return nueva;
9  }
```

Como se ve, luego de pedir la memoria la misma se valida, si la misma fallara la función completa va a fallar devolviendo `NULL`, no hay nada que esta función podría hacer si no tuviera la memoria adecuada. Ahora bien, nuestra función en cierta medida *hereda* el comportamiento de `malloc()`. Si nuestra función hace lo que tiene que hacer devuelve un puntero a un bloque de memoria del heap y si falla devuelve un puntero inválido. Es decir, el que invoque esta función va a tener que tener las mismas precauciones que si invocara a `malloc()`. Por ejemplo:

```
1  int main(void) {
2      char *s = cadena_a_mayusculas("hola");
3      if(s == NULL)
4          return 1;
5
6      printf("%s\n", s);
7
8      free(s);
9      return 0;
10 }
```

Cuando tengamos funciones que manipulan memoria dinámica vamos a tener que ser conscientes todo el tiempo de si las mismas pueden fallar y de liberar los recursos.

Volviendo a la función `free()` la misma recibe de parámetro un puntero devuelvo previamente por `malloc()`. No un puntero al bloque, **el mismo** puntero que devolvió `malloc()`. La función no libera "el puntero", entendiendo por eso a la variable, sino la memoria referenciada

[3]Salvo algunos ejemplos en este mismo capítulo, donde estamos más interesados en explicar la operatoria que en en hacer código consistente y elegante.

por ese puntero, la variable vive en el stack. Por más que ahora empecemos a poner estructuras cada vez más complejas en el heap, las variables que las referencien siempre estarán en el stack.

11.4. Pérdidas de memoria

¿Qué tienen en común estos dos bloques?

```
1 int *p = malloc(10 * sizeof(int));
2 p = malloc(20 * sizeof(int));
3
4 char *s = malloc(5 * sizeof(char));
5 s = "hola";
```

En ambos casos asigno memoria devuelta por `malloc()` a un puntero y acto seguido piso el valor del puntero. ¿Cuál sería la consecuencia de esto? La memoria quedó en un estado inaccesible. Es decir, no puedo recuperar el puntero adonde tengo mis bytes pero tampoco puedo liberarla porque para llamar a `free()` debería haber conservado ese puntero.

Cuando pasa eso se dice que hubo una pérdida de memoria, o un *memory leak*. Y es un error grave en mi programa, porque si mi programa repetidas veces tuviera pérdidas de memoria eventualmente podría consumir todos los recursos de la computadora sin manera de recuperarse.

11.5. Valgrind

A diferencia de otros problemas, identificar memoria no liberada o pérdidas de memoria es muy dificultuoso dado que, a menos que caigamos en un caso extremo donde agotemos toda la memoria, es una falla invisible mirando el comportamiento del programa. Para detectar estos problemas existen herramientas especializadas como Valgrind.

Antes de entrar en Valgrind volvamos a GCC. Cuando nosotros compilamos convertimos código fuente en código máquina, y eso es lo que contiene el ejecutable. Ahora bien, si queremos depurar nuestro ejecutable de poco nos sirve saber en qué instrucción de código máquina hay una falla. Querríamos saber a qué instrucción de código fuente se corresponde cada una de las instrucciones de código máquina. Cuando compilamos podemos habilitar que el compilador incluya información de *debugging* en nuestro ejecutable, que es básicamente esto, incluir la relación entre fuente y compilación. Para ello en GCC podemos agregar el parámetro `-g` en la línea de compilación.

Supongamos el siguiente código:

codigo.c

```
1 #include <stdlib.h>
2
3 int main(void) {
4     char *v = malloc(10);
5     return 0;
6 }
```

Si compilamos y ejecutamos:

```
$ gcc -g codigo.c -o codigo
$ ./codigo
$
```

efectivamente no veremos ningún error, el programa funciona correctamente. Sin embargo nosotros sabemos que olvidamos liberar la memoria pedida.

Si ejecutamos nuestro programa a través de Valgrind en cambio obtendremos:

```
$ valgrind ./codigo
==10746== Memcheck, a memory error detector
==10746== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==10746== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==10746== Command: ./codigo
==10746==
==10746==
==10746== HEAP SUMMARY:
==10746==     in use at exit: 10 bytes in 1 blocks
==10746==   total heap usage: 1 allocs, 0 frees, 10 bytes allocated
==10746==
==10746== LEAK SUMMARY:
==10746==    definitely lost: 10 bytes in 1 blocks
==10746==    indirectly lost: 0 bytes in 0 blocks
==10746==      possibly lost: 0 bytes in 0 blocks
==10746==    still reachable: 0 bytes in 0 blocks
==10746==         suppressed: 0 bytes in 0 blocks
==10746== Rerun with --leak-check=full to see details of leaked memory
==10746==
==10746== For counts of detected and suppressed errors, rerun with: -v
==10746== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
$
```

Focalicémosnos en el "heap summary" (resumen del heap) dice "en uso a la salida 10 bytes en 1 bloque", además me dice que hice un `malloc()` pero ningún `free()`. Más adelante está el resumen de pérdidas, que me dice que perdí de forma definitiva 10 bytes en un bloque.

Casi al final nos da una sugerencia, nos dice que volvamos a correr Valgrind pero esta vez agregando `--leak-check=full`. Si lo hiciéramos obtdríamos (la misma salida de antes pero además):

```
$ valgrind --leak-check=full ./codigo
...
==10799== 10 bytes in 1 blocks are definitely lost in loss record 1 of 1
==10799==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd6
                                                                    4-linux.so)
==10799==    by 0x10865B: main (codigo.c:4)
...
$
```

Este nuevo bloque nos dice que el pedido de memoria fue en la función `main()` y más específicamente en `codigo.c:4`, es decir, en la línea 4 del archivo `codigo.c`[4].

Valgrind no nos va a decir dónde deberíamos haber liberado la memoria, eso es una responsabilidad nuestra como programadores. Lo que Valgrind nos va a inicar es dónde se pidió la memoria que luego no se liberó.

Agreguemos el `free()` en la línea previa al **`return`** y corramos de nuevo:

```
$ valgrind --leak-check=full ./codigo
...
```

[4]Si no hubiéramos compilado con `-g` veríamos sólo que tenemos un error en `0x10865B`, la posición en el código máquina: Compilá con `-g`.

```
==11032== HEAP SUMMARY:
==11032==     in use at exit: 0 bytes in 0 blocks
==11032==   total heap usage: 1 allocs, 1 frees, 10 bytes allocated
==11032==
==11032== All heap blocks were freed -- no leaks are possible
...
$
```

Además de que el resumen nos dice que la cantidad de pedidos y liberaciones es la misma, textualmente nos dice "Todos los bloques del heap fueron liberados – no hay pérdidas posibles". Ese es el mensaje que esperamos obtener que marca la validez de nuestro programa.

Más allá de que el uso primario de Valgrind sea para detectar memoria no liberada o perdida, Valgrind también nos va a avisar cuando utilicemos memoria no inicializada, escribamos en memoria que no nos pertenece, etc.

11.6. `realloc()`

Supongamos el problema de leer números de `stdin` hasta alcanzar la marca de final de archivo. Este es un problema en el cual al momento de empezar desconocemos el tamaño final que vamos a tener. En estos casos lo que hay que hacer es variar el tamaño de la memoria conforme avancemos con la solución.

Supongamos que tenemos una determinada memoria inicial pedida:

```
1 char *p = malloc(5);
```

y que la previsión de 5 caracteres se quedó corta y queremos extender la memoria de `p` a 10. Siempre podemos hacer:

```
1 char *aux = malloc(10);
2 if(aux == NULL)
3     // Fallé, no hay nada que pueda hacer
4 memcpy(aux, p, 5);
5 free(p);
6 p = aux;
```

Alguien puede decir "pero eso no redimensiona el tamaño de `p`, simplemente se creó un nuevo bloque de mayor tamaño y se apuntó a `p`". Exactamente. Si miro `p` antes del procedimiento tenía 5 bytes, después tiene 10, el contenido que estaba en esos 5 bytes está preservado en el nuevo bloque. **Eso** es redimensionar la memoria.

Este algoritmo exacto está implementado en la función `realloc()`. La función `realloc()` recibe un puntero a un bloque de memoria dinámica y un nuevo tamaño. Se encarga de hacer el pedido de nueva memoria, rescatar el contenido y liberar el bloque recibido. En caso de no poder pedir la memoria nueva simplemente devuelve `NULL` y no modifica el bloque recibido.

Un ejemplo de uso sería:

```
1 char *p = malloc(5);
2 ...
3 char *aux = realloc(p, 10);
4 if(aux == NULL)
5     // No pude redimensionar
6 else
7     p = aux;
8 ...
9 free(p);
```

Nunca debe hacerse `p = realloc(p, n);`, dado que si no se pudiera generar la nueva memoria la función devolvería `NULL` y perderíamos la referencia a `p` teniendo una fuga de memoria. Siempre llamaremos a `realloc()` utilizando un puntero auxiliar.

Notar que el tamaño que le pasamos a `realloc()` es el nuevo tamaño deseado. El mismo puede ser tanto mayor como menor que el tamaño previo. La función realizará el `memcpy` por el mínimo entre la cantidad de bytes del tamaño viejo y el nuevo.

11.7. Casos de borde

Un par de casos de borde contemplados en el estándar que hacen más sencilla la implementación de algoritmos:

- `malloc(0)` $\Longrightarrow$ `NULL`.
- `free(NULL)` $\Longrightarrow$ No hace nada.
- `realloc(NULL, n)` $\Longleftrightarrow$ `malloc(n)`.
- `realloc(p, 0)` $\Longleftrightarrow$ `free(p)`.

Por ejemplo:

```
1  /* Lee enteros de stdin hasta agotar la entrada, devuelve el
2  vector de enteros leídos por el nombre y la cantidad de enteros
3  a través de n. */
4  int *leer_enteros(size_t *n) {
5      int *v = NULL;
6      *n = 0;
7
8      char buffer[100];
9      while(fgets(buffer, 100, stdin) != NULL) {
10         int *aux = realloc(v, (*n + 1) * sizeof(int));
11         if(aux == NULL) {
12             free(v);
13             return NULL;
14         }
15         v = aux;
16         v[(*n)++] = atoi(buffer);
17     }
18
19     return v;
20 }
```

(¿Es buena idea descartar todo lo leído si no hay memoria? Es un criterio, tal vez no sea el mejor.)

El código anterior utiliza dos de los casos de borde que se mencionaron. Te queda de tarea identificar cuáles y desarrollar cómo debería haber sido el código en caso de que el estándar no especificara que esos casos son seguros.

Sobre el código anterior notar que redimensionamos por cada línea leída, esto fuerza un llamado a `malloc()` y a `memcpy()` por línea leída... donde a medida que se avanza hay que copiar más y más y más. Profundizaremos más sobre estos aspectos de eficiencia más adelante, pero esta solución es muy ineficiente. Necesita copiar una cantidad de bytes proporcional al cuadrado de la cantidad de líneas leídas.

11.8. Matrices dinámicas

Siguiendo con lo ya presentado en la sección 8.10 si quisiéramos crear una matriz *en el sentido* de las matrices estáticas de C podríamos hacer algo como:

```
1 float (*matriz)[cols] = malloc(filas * sizeof(float [cols]));
2 ...
3 matriz[i][j] = 2.5;
```

Si hiciéramos esto tendríamos toda la memoria en un bloque monolítico de `filas * cols * sizeof(float)` bytes y cuando accedemos con el doble corchete lo hacemos en el sentido unidimensional que ya vimos, como un equivalente a `*(matriz + i * cols + filas)`.

Si bien se puede hacer esto, este no es el enfoque preferido cuando utilizamos memoria dinámica. Con memoria dinámica efectivamente podemos generar un arreglo dinámico de arreglos dinámicos, donde efectivamente haya una bidimensionalidad.

El enfoque dinámico es (se omiten las validaciones):

```
1 float **matriz = malloc(filas * sizeof(float*));
2 for(size_t f = 0; f < filas; f++)
3     matriz[f] = malloc(cols * sizeof(float));
4 ...
5 matriz[i][j] = 2.5;
```

En este caso `matriz` es un arreglo de punteros a flotantes, cada uno de sus elementos será un puntero a flotante. Ahora bien, puntero a flotante es el tipo de los arreglos dinámicos, dado que puedo acceder a un arreglo a través de un puntero a su primer elemento. Luego cada uno de los elementos de la matriz, un `matriz[i]` será un puntero a flotante y lo inicializaremos con un vector dinámico.

En este caso notar que los dobles corchetes efectivamente significan lo esperado `*(*(matriz + i)+ j)`, acá no hay magia de la que C hace para resolver las matrices estáticas. Dos asteriscos significan desreferenciar punteros dos veces.

El enfoque de arreglos dinámicos de arreglos dinámicos consume un poco más de memoria, porque ahora se necesita un arreglo de `filas` punteros, pero es mucho más flexible al mantener la memoria segmentada en bloques de no más de `cols` elementos consecutivos.

Capítulo 12

Contratos

12.1. Documentación

Hasta el momento en la sección 2.8 introdujimos el concepto de comentarios y su sintaxis y los hemos utilizado en muchos de los ejemplos que desarrollamos. Ahora bien nunca formalizamos qué es un comentario y qué es la documentación.

Los comentarios son lo que hemos estado utilizando hasta el momento: Explicaciones al margen del código que ayudan al que está leyendo el código fuente a entender **cómo** se está haciendo algo o **por qué** se decidió hacerlo de esa manera. El destinatario de esos comentarios es un programador[1] que está queriendo mantener el código desarrollado. Cabe destacar, quien lee código fuente sabe programar, por lo tanto los comentarios no deben explicar por ejemplo que `int i = 0;` declara una variable de tipo entero que se llama `i` y le asigna el literal 0. Tal vez (y sólo tal vez) tenga sentido explicar qué va a representar `i` en el código. Los comentarios ayudan a entender cosas que no son obvias o consideraciones que no son evidentes.

Si bien los comentarios constituyen parte de la documentación del código vamos a jerarquizar el concepto de documentación de otra manera: Explicaciones de **qué** hace el código, generalmente funciones o módulos. El destinatario de la documentación es aquel que quiera **utilizar**, por ejemplo, nuestra función. Al que quiera utilizar una función le interesa saber de qué forma se utiliza, qué representan los parámetros, qué devuelve, etc. y probablemente tenga poco interés en saber cómo se implementó. Un ejemplo de documentación es la que está asociada a todas las funciones de biblioteca que presentamos hasta el momento. Quien lee documentación quiere, por ejemplo, saber cómo invocar a la función `printf()`, o cómo imprimir un número de punto flotante con 5 decimales, pero no tiene el menor interés en saber cómo es que `printf()` hace su magia. La documentación no explica el código fuente, es más, muchas veces ni siquiera tenemos acceso a él, la documentación explica cómo interactuar con las interfases.

Por más que muchas veces sea descuidada en un proyecto, la documentación es parte importante del mismo. Un buen código puede ser muy expresivo, pero nunca explica por sí solo las decisiones de diseño, los casos particulares analizados, los algoritmos utilizados, etc.

12.2. Autodocumentación

Si bien documentar es importante, la documentación empieza por el código en sí. Imaginemos que tenemos esta función:

[1] Notar que no decimos *"otro programador"*, tranquilamente podemos estar dejando un comentario para nosotros mismos en el futuro.

```
1 void f(int *a, int *b, int c);
```

La pregunta es, ¿qué hace la función? Si la función tiene una documentación podemos leerla, y tal vez después de eso podamos saber para qué servía.

En realidad esta función ya la implementamos en la sección 8.7.1. La firma de la función fue

```
1 void copiar_vint(int destino[], const int origen[], size_t n);
```

¿Mejora en algo haber elegido un nombre claro para la función, para los parámetros, haber dejado en claro que los primeros parámetros en realidad son vectores, que el segundo de ellos es constante, que el tercer parámetro no es cualquier entero si no un tamaño de memoria? La idea de la autodocumentación es que si somos claros en la intención de nuestras funciones entonces no tenemos una necesidad de leer la documentación para entender qué hace una función.

¿Esto significa que podemos prescindir de la documentación? No, todavía podemos querer explicar detalles al respecto de la misma. Lo que esto significa es que si estamos leyendo un fragmento de código fuente donde se llama a una función `copiar_vint` probablemente no haga falta ir a la documentación para saber que esa función copia arreglos de enteros. Del mismo modo que si quisiéramos usar la función no tendríamos que preguntarnos cuál de los dos parámetros es el destino y perder el tiempo leyendo texto.

Y, dicho sea de paso, la diferencia más importante entre una función y la documentación es que la función se compila y se ejecuta. Es decir, la función `copiar_vint` es exactamente lo que está programado. La documentación puede contener errores o incluso estar desactualizada si el proyecto se trató de forma negligente. Escribir código más claro implica poder documentar de forma más compacta y no tener que mantener tanta documentación.[2]

La autodocumentación tiene limitaciones, pero es el comienzo para tener un código legible.

12.3. Contratos

La programación por contratos es una formalización del concepto de documentación que intenta separar responsabilidades al respecto de de qué son responsables no sólo el que implementó la función si no el que va a invocarla. La idea de la práctica de la programación por contratos es dejar estipuladas por escrito las reglas de uso y el comportamiento.

El contrato no es otra cosa que documentación, pero programar por contratos es dejar en claro en esa documentación determinadas responsabilidades:

Precondiciones: Las precondiciones de un contrato son condiciones que deben cumplirse antes de ejecutar la función, es decir, son responsabilidad del que la invoca. Estas pueden ser referidas a los valores recibidos, a condiciones sobre esos valores, etc. Por ejemplo, para la función `copiar_vint` dada en la sección previa son precondiciones que tanto el vector `origen` como `destino` existan y tengan al menos `n` elementos cada uno.

¿Qué implica establecer una precondición?, que no es responsabilidad de la función validar o considerar qué pasaría si los datos de origen no son correctos. Y si bien en otros lenguajes esto es un tema de comodidad y de focalizar las validaciones en donde realmente sean necesarias, en el lenguaje C muchas veces ni aunque pudiéramos podríamos realizar esa validación. El ejemplo de la función que copia enteros es más que válido, ¿cómo podría saber dentro de la función que los tamaños son adecuados si la función sólo recibió punteros?

[2] Por cierto, de forma irónica en programación se suele decir *"la documentación está* **en** *el fuente"*, para justificar que un código sin nada de documentación todavía puede entenderse simplemente leyéndolo.

Postcondiciones: Las postcondiciones son condiciones que deben cumplirse luego de ejecutar la función, es decir, son responsabilidades del que implementa la función. Ahora bien hay un pero importante: Sólo se podrán garantizar las postcondiciones si las precondiciones se cumplieron. Es decir, el que implementa la función se compromete a hacer algo siempre y cuando sea llamado de manera correcta.

Por ejemplo, en el ejemplo ya dado, la función se compromete a que los `n` elementos del vector `origen` sean copiados en el vector `destino`... ahora bien, sólo si los vectores existen y son del tamaño adecuado.

Si bien las precondiciones y postcondiciones se pueden escribir de forma explícita en la documentación (es decir "Precondiciones: ...") también pueden ser integradas al texto de las mismas de forma implícita. Lo importante de la documentación es que quede en claro qué se espera recibir en cada parámetro y qué se devolverá o modificará luego de haber sido invocado correctamente.

12.4. `assert()`

Tanto las precondiciones como las postcondiciones entran dentro de lo que se denominan "aseveraciones" (o *"assertions"* en inglés). Las aseveraciones son expresiones que tienen que ser válidas siempre.

En el lenguaje C se provee un encabezado, `<assert.h>`, que contiene una macro `assert()` la cual recibe una expresión booleana. La intención de `assert()` es aseverar que esa expresión sea cierta. Por ejemplo:

test.c

```
1  #include <assert.h>
2
3  /*
4  Recibe dos enteros numerador y denominador y calcula la división.
5  Precondiciones: El denominador debe ser distinto de cero.
6  Postcondiciones: Devuelve el cociente entre numerador y
       ↪ denominador.
7  */
8  int dividir(int numerador, int denominador) {
9      assert(denominador != 0);
10     return numerador / denominador;
11 }
```

Si la expresión evaluara a `true` `assert()` no hará nada. Ahora bien, si llegara a evaluar a `false`, que en este caso está asociado con haber violado una precondición, entonces el programa se abortará indicando dónde se violó la aseveración. Por ejemplo:

```
$ ./test
test: test.c:19: dividir: Assertion 'denominador != 0' failed.
$
```

La macro `assert()` es una herramienta para realizar pruebas cuando se desarrolla código, porque justamente permite testear cuando algo no está evaluando a lo que se espera.

Ahora bien, un programa en producción que se aborta cada vez que no se verifica una aseveración puede ser peligroso dependiendo de la aplicación. Por ejemplo, podríamos aseverar que una división no puede ser por cero o que no puede llamarse a una raíz con un valor negativo, pero si mi aplicación fuera un juego 3D que realiza millones de evaluaciones de

distancias por segundo el resultado de una división por cero tal vez sea un pixel mal visto. Si hubiera una cuenta inválida, ¿prefiero que se cierre el juego o que haya un error de visualización indetectable? Entendamos que en la etapa de desarrollo querría erradicar cualquier fragmento que opere de forma anómala, pero cuando el código está siendo utilizado por el usuario esto ya no es positivo.

La particularidad que tienen las macros de `assert()` es que pueden ser eliminadas del código. Invocando al compilador con el parámetro `-DNDEBUG` se inhabilitan todas las aseveraciones. Para el ejemplo anterior:

```
$ gcc test.c -o test -DNDEBUG
$ ./test
Excepción de coma flotante
$
```

(Nadie dijo que deshabilitar los asserts iba a hacer que el código funcionara.)

12.5. Invariantes de ciclo

En programación las "invariantes" son condiciones que no se modifican a lo largo de la ejecución. Una invariante de ciclo es una condición que siempre se verificará al comienzo de cada ciclo de una iteración.

Ejemplo, si tenemos el código:

```
1 int maximo(const int v[], size_t n) {
2     int max = v[0];
3     for(size_t i = 1; i < n; i++)
4         if(v[i] > max)
5             max = v[i];
6     return max;
7 }
```

podemos definir como invariante que `max` siempre va a tener el máximo del arreglo en el rango $[0 \,..\, i)$.

¿Para qué nos sirven las invariantes de ciclo? En principio son una definición formal que sirve para demostrar algoritmos, pero en la práctica nos pueden ayudar a ordenar cómo encaramos nuestros algoritmos. Por ejemplo, si estuviéramos desarrollando el código, ¿cómo fue que tomamos la decisión de en la línea 2 inicializar `max = v[0]`? Notar que esa inicialización es inmediata mirando nuestra invariante: Si `max` debe contener siempre el máximo del arreglo hasta `i` entonces en la primera iteración, tiene que contener `v[0]`.

12.6. Alan y Bárbara

Cuando se presentaron las precondiciones y postcondiciones se habló del "que implementa" y del "que invoca" una función. A partir de ahora nos vamos a referir mucho a esos dos roles y es preferible ponerles un nombre. Los nombres podrían ser el usuario A y el usuario B, pero preferimos utilizar los nombres Alan y Bárbara en honor a los programadores Alan Turing y Bárbara Liskov.

Entonces, cuando hablemos de implementaciones vamos a tener siempre a nuestros dos personajes:

Alan: El programador que implementa una función, o módulo o bloque.

Bárbara: El programador que utiliza esa función, o módulo o bloque.

Cabe destacar que estos personajes no son personas sino que son roles, es decir, cuando nos refiramos a que Alan hace determinada cosa y Bárbara hace determinada otra, no estamos diciendo que tenga que haber dos programadores involucrados. Es perfectamente esperable que un programador cuando desarrolle una función cumpla el rol de Alan y que cuando utilice esa misma función cumpla el rol de Bárbara, no hace falta que haya terceros involucrados.

Del mismo modo los roles de Alan y Bárbara son relativos. Si Alan implementa una función, y esa función, por ejemplo, utiliza la función `sqrt()` desde el punto de vista de `sqrt()` el rol que llamábamos Alan ahora será Bárbara. Siempre vamos a ser el Alan o la Bárbara de alguien más.

Capítulo 13

Tipo de Dato Abstracto

Tal vez lo hayas notado pero en el último capítulo no introdujimos nada nuevo del lenguaje C, esto es porque ya prácticamente cubrimos todo lo que tiene el lenguaje. Sin embargo apenas estamos promediando el curso. A partir de ahora lo que vamos a hacer es profundizar por un lado en algoritmos y por el otro en construir programas complejos.

Cuando hablamos de programas complejos no hablamos de códigos de 1000 líneas. Un programa complejo puede tener cientos de veces esa cantidad. Por ejemplo en el año 2020 Mozilla Firefox tenía 21 millones de líneas, en el 2021 Linux tenía 30 millones de líneas, mientras que Chromium tenía cerca de 35 millones de líneas de código. La pregunta es cómo podemos hacer para mantener organizado código con esa extensión.

13.1. Tipo de Dato Abstracto

Hasta el momento cada vez que pusimos cosas en la memoria lo hicimos conociendo la organización de esa memoria. Por ejemplo, en la sección 11.8 vimos dos formas diferentes de poner en memoria una matriz, del mismo modo que en la sección 9.1 vimos diversos ejemplos de empaquetamiento de datos en estructuras. Estos tipos son lo que se llaman tipos concretos. Para utilizar el tipo necesito conocer cómo es su estructura.

Ahora bien este enfoque de conocer cómo están estructuradas las cosas por dentro no permiten escalar en complejidad. Así como cuando utilizamos una función no queremos saber cómo es que hace lo que hace (por ejemplo, ¿cómo hace `printf()` para mostrar algo por la pantalla?) queremos extender ese comportamiento también a las estructuras de datos. Si podemos encapsular los datos de tal manera que no tengamos que preocuparnos por cómo están implementados y sólamente con utilizarlos para construir nuestros programas, podemos compartimentar los distintos bloques de un proyecto de tal manera que sea viable construirlo combinando estos bloques para construir bloques más abstractos y de más alto nivel.

Si podemos estructurar un tipo detrás de una interfaz basada en llamadas a funciones, entonces podemos prescindir de conocer cómo está representado internamente ese tipo y de cómo manipularlo. Cuando encapsulamos un tipo nos interesa qué y no cómo lo hace. Este encapsulamiento es lo que se conoce como Tipo de Dato Abstracto (TDA).

En la concepción del TDA queremos que Alan pueda implementar completo el tipo conociendo todos sus detalles y que Bárbara pueda utilizarlo de forma completamente opaca sin preocuparse por la estructura o por la complejidad del trabajo de Alan. En nuestro modelo no sólo a Bárbara no le importa saber cómo lo hizo Alan, vamos a llevarlo más allá y vamos a impedir que Bárbara sepa cómo está hecho. Por eso se dice que el tipo es abstracto, desde la perspectiva de Bárbara hace cosas pero no expone cómo las hace.

13.2. Interfaz

Un TDA va a estar formado por dos cosas, uno es el tipo en sí, que puede ser lo que más cómodo le quede a Alan, una estructura, un arreglo, un entero, etc. sobre el cual Bárbara va a ser totalmente ignorante al respecto de cómo se guarda la información dentro de él. La otra cosa es una necesidad, como Bárbara no tiene acceso al tipo, Alan va a tener que proveer todas las funciones que haga falta según las cosas que se puedan hacer con el tipo.

Entonces un TDA es un tipo y un conjunto de funciones, que llamaremos primitivas.

Imaginemos que tenemos un tipo `complejo_t` que representa a los números complejos $\mathbb{C}$. Las primitivas que me provea el tipo tendrán que ser consecuentes con lo que espero hacer con complejos, por ejemplo sumarlos, restarlos, conjugarlos, etc. Digamos que una interfaz del tipo podría ser:

```
1 complejo_t *complejo_sumar(const complejo_t *a, const complejo_t *
    ↪ b);
2 complejo_t *complejo_restar(const complejo_t *a, const complejo_t
    ↪ *b);
3 complejo_t *complejo_multiplicar(const complejo_t *a, const
    ↪ complejo_t *b);
4 complejo_t *complejo_dividir(const complejo_t *a, const complejo_t
    ↪  *b);
5
6 complejo_t *complejo_conjugar(const complejo_t *a);
7 complejo_t *complejo_inverso(const complejo_t *a);
```

Como ya dijimos, estas funciones se llaman "primitivas".

Ahora bien, si sólo tuviéramos estas primitivas nuestro tipo sería inútil. Si Bárbara no sabe cómo está representado el tipo internamente, ¿para qué le sirve hacer operaciones si no puede saber el resultado? Es decir, tengo dos complejos, los sumo, eso me genera un nuevo complejo. ¿Cuánto vale ese complejo? Incluso, yendo más allá, ¿de dónde saco los dos complejos que quiero sumar?

El formato del TDA obliga a que, por fuera de las primitivas que necesito para operar mi tipo, haya primitivas que sirven únicamente para gestionar el tipo.

13.2.1. Constructores y destructores

Para empezar a usar el TDA necesito primero alguna primitiva que me devuelva una instancia de dicho TDA a partir de datos que no pertenezcan al TDA. En nuestro ejemplo de complejos necesitamos poder generar un complejo desde una parte real e imaginaria, o desde un módulo y un argumento o sencillamente tener una primitiva que me devuelva el complejo 0, o el complejo 1 o el complejo i:

```
1 complejo_t *complejo_crear_ri(float real, float imaginaria);
2 complejo_t *complejo_crear_ma(float modulo, float argumento);
3 complejo_t *complejo_cero();
```

Las primitivas que crean TDAs se llaman "constructores".

Del mismo modo que el TDA se crea el TDA tiene que poder destruirse y al desconocer Bárbara qué contiene dentro, tiene que delegar en Alan esta operación:

```
1 void complejo_destruir(complejo_t *c);
```

La primitiva que destruye un TDA se llama "destructor".

Notar que todas las primitivas del tipo tienen de prefijo el nombre del tipo `complejo_`. Esta es una manera de asociar las primitivas al tipo en cuestión.

13.2.2. Getters y setters

Con los constructores y destructores resolví la primera parte de mi problema, que era generar complejos para empezar a operar. Pero de qué me sirve sumar el complejo devuelto por `complejo_crear(1, 2)` con `complejo_crear(0, 1)` si el resultado es otro complejo que es opaco para mí.

El complejo tiene que tener una forma de extraer su contenido a tipos de datos externos al TDA, por ejemplo:

```
1 float complejo_real(const complejo_t *c);
2 float complejo_imaginaria(const complejo_t *c);
3 float complejo_modulo(const complejo_t *c);
4 float complejo_argumento(const complejo_t *c);
```

Las primitivas que me dejan obtener datos internos del TDA se llaman "getters".

Del mismo modo, puedo tener primitivas que me dejen escribir un dato interno del TDA, como por ejemplo, darle un valor determinado a la parte real o a la imaginaria. Estas primitivas se denominan "setters".

El tipo, los constructores, destructores, getters, setters y primitivas en general, constituyen la interfaz del tipo. Esto más la documentación[1] constituirá el contrato del TDA que implementa Alan y consume Bárbara.

13.3. Bárbara

Con lo presentado hasta el momento, Bárbara ya tiene todo lo necesario como para implementar los cálculos que necesite.

Cabe hacer la aclaración de que la interfaz elegida tal vez no sea la más práctica para implementar un TDA pero sí la más similar a cómo se implementan tipos más... *complejos*. Bárbara podría hacer algo como:

```
1  complejo_t *a = complejo_crear_ri(1, 2);
2  complejo_t *b = complejo_crear_ri(0, 1);
3
4  complejo_t *r = complejo_sumar(a, b);
5
6  printf("Real: %f, Imaginaria: %f\n", complejo_real(r),
       ↪ complejo_imaginaria(r));
7
8  complejo_destruir(a);
9  complejo_destruir(b);
10 complejo_destruir(c);
```

(Sí, tenemos una Bárbara que no se preocupa por si las cosas pueden fallar.)

Notar que implementamos la parte de Bárbara basados pura y exclusivamente en el contrato. No necesitamos nada más. El nivel de abstracción es tal que todavía ni siquiera discutimos cómo es que Alan va a implementar lo que tiene que implementar. No hace falta, si conocemos la interfaz del tipo podemos usarlo incluso aunque Alan ni haya empezado a diseñarlo.

[1] En este caso la documentación es volcar lo que explicamos cuando fundamentamos cada una de estas funciones.

13.4. Alan

Si bien para Bárbara el TDA es un tipo abstracto, para Alan será un tipo concreto. La potencia del paradigma es que Alan puede definir el tipo concreto que quiera, cambiarlo más adelante, o incluso puede venir otro Alan a proveer una implementación diferente del mismo tipo.

Siempre y cuando la representación que Alan elija sirva para poder resolver los casos de uso de las primitivas, Alan puede elegir la representación interna que quiera. Incluso en un ejemplo tan sencillo como este hay múltiples formas de crear el tipo. Por ejemplo:

```
// El tipo es un arreglo de dos elementos (¿qué es cada uno?):
typedef float complejo_t[2];

// El tipo es una estructura:
typedef struct {
    float real, imaginaria;
} complejo_t;

// El tipo es otra estructura:
typedef struct {
    float modulo, argumento;
} complejo_t;

// Empaquetamos 2 floats de 32 bits en un entero de 64 bits:
typedef uint64_t complejo_t;
```

Cada representación tendrá ventajas y desventajas, algunas primitivas serán más fáciles de implementar, otras más difíciles.

Supongamos que implementamos la estructura con parte real e imaginaria, veamos algunas primitivas:

```
// Un constructor:
complejo_t *complejo_crear_ri(float real, float imaginaria) {
    complejo_t *c = malloc(sizeof(complejo_t));
    if(c == NULL) return NULL;

    c->real = real;
    c->imaginaria = imaginaria;

    return c;
}

// El destructor:
void complejo_destruir(complejo_t *c) {
    free(c);
}

// Un getter:
float complejo_real(const complejo_t *c) {
    return c->real;
}

// Una primitiva genérica:
```

```
23 complejo_t *complejo_sumar(const complejo_t *a, const complejo_t *
       ↪ b) {
24     return complejo_crear_ri(a->real + b->real, a->imaginaria + b
           ↪ ->imaginaria);
25 }
```

Más allá de que se haya elegido un ejemplo sencillo, suele ser común que las primitivas de un TDA sean funciones sencillas y de pocas líneas. Como el tipo va a ser utilizado por Bárbara, entonces la lógica de la interfaz tiende a implementar operaciones atómicas y muy concretas, además de que el diseño que se hace tiende a ordenar cómo se resuelven las cosas. Por lejos de complicar las cosas los TDAs las simplifican y estructuran, tanto para Alan como para Bárbara.

Como un apunte al margen se vuelve al comentario de que tal vez esta no sea la mejor interfaz para este TDA. ¿Por qué?, porque un complejo es un tipo muy sencillo que apenas tiene dos valores y hacer que cada operación devuelva una nueva instancia creada con memoria dinámica genera mucha sobrecarga en nuestro código. Tal vez un tipo tan *sencillo* como el *complejo* podría resolverse con un intermedio entre los tipos concretos y los abstractos donde las funciones tomen y devuelvan estructuras sin utilizar punteros y Bárbara conozca cómo es la implementación interna[2] En ese caso no tendríamos destructores, ni tendríamos que validar memoria, aunque probablemente sería cómodo tener constructores que armen el tipo.

13.5. Invariantes de representación

En la sección 12.5 se definieron las invariantes. En los TDAs existen las invariantes de representación, que van a ser muy importantes en nuestro diseño. Al igual que otras invariantes, las invariantes de representación son condiciones que van a ser siempre ciertas en la representación interna de los tipos.

La invariante de representación de un determinado TDA será algo interno de Alan y su implementación. Alan se encargará de definir qué cosas quiere definir como invariante.

¿Para qué sirven? La idea es esta, si las invariantes tienen que cumplirse siempre y Alan es el único que puede manipular los datos del tipo, entonces la invariante de representación será a su vez precondición y postcondición de todas las primitivas. Es decir, como Alan va a garantizar que ninguna primitiva va a romper la invariante, entonces puede tener asegurado que el tipo va a venir siempre con la invariante correcta.

Volvamos al ejemplo de los complejos, pero en este caso Alan definió su representación interna con `struct { float modulo, argumento; };` y tiene que implementar una primitiva para comparar dos complejos. Por ejemplo:

```
1 bool complejo_son_iguales(const complejo_t *a, const complejo_t *b
      ↪ ) {
2     return a->modulo == b->modulo && a->argumento == b->argumento;
3 }
```

¿Está bien implementada esta primitiva? Si tengo el complejo $0\angle 0$, ¿es el mismo o no que el complejo $0\angle\pi$? ¿Y el complejo $1\angle 0$ es o no el mismo que $1\angle 2\pi$? ¿Y el complejo $1\angle\pi$ es o no el mismo que $-1\angle 0$?

La primitiva anterior no funciona. Y hacer funcionar a esa primitiva es complicado cuando comparar dos complejos debería ser una tarea sencilla.

¿Cambiaría algo si garantizáramos invariantes sobre la representación? Propongamos esto:

```
1 typedef struct {
2     /* Representa a un número complejo en su forma polar.
```

[2] Vamos a explicar a qué nos referimos un poco más adelante.

```
3         Invariante de representación:
4          - modulo >= 0
5          - 0 <= argumento < 2pi
6          - Si modulo = 0 => argumento = 0 */
7     float modulo, argumento;
8 } complejo_t;
```

Si la invariante fuera esa, entonces la implementación de `complejo_son_iguales()` ya dada sería correcta y **principalmente** sencilla. ¿Todo el código sería sencillo? No particularmente, por ejemplo, el constructor `complejo_crear_ma()` tendrá que validar y ajustar los parámetros recibidos por Bárbara quién **no** conoce **ni** debe conocer la invariante de Alan. Ahora bien, esto es algo que se hace únicamente al modificar o recalcular módulos y argumentos, luego simplifica el resto de las primitivas dado que podremos contar con que los complejos tendrán una representación única.

Salgamos del ejemplo de los complejos y pensemos por ejemplo en un vector dinámico que puede almacenar elementos enteros de a uno por vez:

```
1 typedef struct {
2     /* Vector dinámico de enteros.
3        Invariante de representación:
4         - v es un vector de n elementos
5         - v == NULL si y sólo si n == 0 */
6     int *v;
7     size_t n;
8 } vector_t;
```

Y lo que se definió como invariante puede parecer poco pero es tremendamente ordenador sobre cómo implementar las primitivas, por ejemplo:

```
1  vector_t *vector_crear() {
2      vector_t *v = malloc(sizeof(vector_t));
3      if(v == NULL) return NULL;
4
5      v->v = NULL;
6      v->n = 0;
7
8      return v;
9  }
10
11 void vector_destruir(vector_t *v) {
12     free(v->v);
13     free(v);
14 }
15
16 bool vector_agregar_elemento(vector_t *v, int elemento) {
17     int *aux = realloc(v->v, (v->n + 1) * sizeof(int));
18     if(aux == NULL) return false;
19
20     v->v = aux;
21     v->v[v->n++] = elemento;
22
23     return true;
24 }
```

¿Podés identificar dónde estamos siendo obligados y dónde estamos usando a nuestro favor la invariante de representación?

Tanto en la inicialización del constructor, como en incrementar el valor de `v->n` después de agregar un elemento estamos haciendo cosas para cumplir la invariante.

Ahora bien, en el destructor estamos liberando `v->v` sin nunca chequear que el vector esté vacío, del mismo modo, en la primitiva de agregar elemento estamos redimensionando el vector sin revisar que esté vacío. Ambas operaciones pueden hacerse con seguridad porque está garantizado el comportamiento de `free()` y `realloc()` cuando el parámetro es `NULL` (ver sección 11.7).

Dicho sea de paso, notar que si falla el agregado de un elemento sólo falla esa operación y nada más. El contenido previo del TDA (y su invariante) se preserva. Ni siquiera una falla grave de memoria debería romper la invariante.

Volviendo a las cosas que pudimos simplificar porque las habíamos definido en la invariante debemos remarcar que sólo pudimos simplificarlas por haberlas definido como invariante. Si no hubiéramos definido ese invariante (o alguno diferente, no es el único posible) no podríamos haber asumido nada al respecto de los datos, porque nadie garantizaría una consistencia en todas las primitivas del tipo. Dicho de otra forma, no podemos asumir nada que no hayamos documentado como invariante, si lo hiciéramos estaríamos modelando mal nuestro tipo.

13.6. Modularización

Volvamos al ejemplo del TDA de números complejos. Tenemos a Alan y Bárbara codificando para un mismo proyecto, ¿pueden desacoplar sus códigos?

En el próximo capítulo trataremos en detalle el tema de modularización, pero para cerrar el ejemplo explicaremos lo básico acá.

La idea es que Alan y Bárbara puedan trabajar por separado. Es más, nosotros hablamos de los roles de Alan y de Bárbara como roles que se dan en simultaneo, pero esto no tiene por qué ser así. La mayor parte de las veces Alan desarrolló un TDA sin siquiera saber las necesidades de Bárbara y tiempo después Bárbara considera que el TDA de Alan es adecuado para resolver su problema y lo utiliza en su proyecto. Así suele ser la dinámica con las bibliotecas que utilizamos.

Repasemos un poco. Habíamos dicho que el contrato era el tipo, la interfaz y la documentación. Notar que eso en lenguaje C corresponde a declaraciones y comentarios, no a definiciones e implementaciones. Si recordamos de cuando vimos el proceso de compilación los archivos de encabezados `.h` contenían justamente definiciones de tipos, etiquetas y prototipos de funciones.

Entonces, en nuestro tipo el contrato va a terminar siendo un archivo `.h` provisto por Alan.

Ahora bien, ¿cómo incluímos el tipo en el encabezado? Si Alan definiera la estructura en este archivo entonces Bárbara conocería cómo es la representación interna de la estructura. Bueno, el lenguaje C tiene mecanismos para resolver esto.

En el lenguaje C es viable declarar una estructura: `struct complejo;` o `typedef struct complejo compejo_t;` sin definirla. Notar que si declaramos una estructura el compilador no puede conocer su `sizeof()` y por lo tanto no puede reservar memoria para variables. Es más, si tenemos solamente la declaración tampoco conoce los miembros de la estructura por lo que no puede utilizarse el operador punto (ni flecha). Si bien desde el punto de vista de Bárbara esto es exactamente lo que necesitamos, cabe hacerse la pregunta de para qué sirve declarar una estructura si no puedo ni declarar variables ni acceder a su contenido. Bueno, la respuesta es que con la declaración puedo declarar punteros. Recordar que un puntero no es otra cosa que una dirección de memoria, para declarar una variable de tipo puntero sólo hay que saber el tamaño de las direcciones de memoria, no importa el tamaño del tipo apuntado.

Entonces, podremos definir un archivo `complejo.h` con el siguiente contenido:

complejo.h

```
1  // ARREGLAME: Falta documentar todo esto.
2  typedef struct complejo complejo_t;
3
4  complejo_t *complejo_crear_ri(float real, float imaginaria);
5  complejo_t *complejo_crear_ma(float modulo, float argumento);
6  complejo_t *complejo_cero();
7
8  void complejo_destruir(complejo_t *c);
9
10 complejo_t *complejo_conjugar(const complejo_t *a);
11 complejo_t *complejo_inverso(const complejo_t *a);
12 float complejo_real(const complejo_t *c);
13 float complejo_imaginaria(const complejo_t *c);
14 float complejo_modulo(const complejo_t *c);
15 float complejo_argumento(const complejo_t *c);
16
17 complejo_t *complejo_sumar(const complejo_t *a, const complejo_t *
       ↪ b);
18 complejo_t *complejo_restar(const complejo_t *a, const complejo_t
       ↪ *b);
19 complejo_t *complejo_multiplicar(const complejo_t *a, const
       ↪ complejo_t *b);
20 complejo_t *complejo_dividir(const complejo_t *a, const complejo_t
       ↪  *b);
```

Luego Bárbara escribirá su "`main.c`" haciendo un **`#include`** de este archivo, y Alan implementará su `complejo.c` el cual definirá la **`struct`** `complejo` y las funciones.

Como ya dijimos, terminaremos de explicar la modularización en el próximo capítulo dedicado específicamente a eso.

Capítulo 14

Modularización

14.1. Proceso de compilación

Ya le dedicamos completo el capítulo 4 a explicar el proceso de compilación del lenguaje C. Refresquemos particularmente que es un proceso en tres etapas:

1. Primero viene la etapa de **preproceso**. Durante esta etapa actúa el preprocesador, que realiza reemplazos, inclusiones, etc. En este paso nuestro código se nutre de archivos de encabezados `.h`. Los archivos de encabezados contienen declaraciones de tipos, funciones, etiquetas, macros, etc.

2. Luego viene la etapa de **compilación**. Durante esa etapa actúa el compilador, que traduce nuestro código fuente en código objeto (que es equivalente al código máquina). Este proceso no toma nada del exterior, es nuestro código complementado con las declaraciones que trajo el preprocesador, pero lo único que es traducible a código objeto es nuestro código fuente.

3. Finalmente viene la etapa de **enlace**. Durante esta etapa actúa el enlazador, que junta nuestro código objeto con el código objeto de las bibliotecas que utilizamos y resuelve las referencias cruzadas que existan. Además el enlazador es el que verifica que haya un único `main()` y lo establece como punto de entrada.

Por diseño C está pensado para compilar programas modulares. Si bien hasta el momento utilizamos el enlazador para juntar el código objeto de nuestro programa con el código objeto de la biblioteca de C, el enlazador puede combinar múltiples códigos objeto para formar un único ejecutable. Entonces podemos partir nuestro proyecto en múltiples archivos `.c`, compilar cada uno individualmente y luego juntar sus códigos objetos para formar un ejecutable.

En la invocación que venimos haciendo del GCC estamos realizando una compilación monolítica, es decir, los tres procesos se ejecutan en secuencia y generan un único ejecutable. Ahora bien, el GCC se puede manipular para ejecutar cada uno de los pasos de forma individual.

En este caso nos interesa compilar, o sea, realizar la etapa de preprocesado y compilación, para generar un código objeto y luego enlazar el código objeto. Si tuviéramos un `fuente.c` podríamos hacer esto:

```
$ gcc fuente.c -c -Wall -std=c99 -pedantic
$ gcc fuente.o -o programa -lm
$
```

La primera línea se encargó de la compilación y generación del código objeto `fuente.o` mientras que la segunda enlazó el código objeto para generar el `programa`. Notar que estamos

distinguiendo qué parámetros son de compilación y cuáles de enlace. No tendría sentido pasarle, por ejemplo, `-lm` a una etapa que no enlaza o `-Wall` a una etapa que no compila.

Así como en la segunda línea enlazamos a `fuente.o` con las bibliotecas `libc.so`[1] y `libm.so` las bibliotecas de C y matemáticas respectivamente, también podemos enlazar contra otros códigos objeto. Es decir, podemos realizar múltiples compilaciones de archivos `.c` con `-c` y luego enlazarlas todas juntas en un único ejecutable. El único requisito es que entre todos los objetos se aporte uno y sólo un `main()`.

14.2. Modularización

Como se adelantó en el capítulo anterior, la modularización consiste en separar proyectos en múltiples archivos `.c` y comunicar esos archivos con archivos `.h`.

En el ejemplo del capítulo anterior desarrollamos un TDA para manejar números complejos e hicimos una aplicación que los utilizaba. En la sección 13.6 anticipamos un borrador de cómo sería la modularización.

Habiamos llegado a que teníamos tres archivos: `main.c`, donde Bárbara implementó sus operaciones con complejos y `complejo.c` y `complejo.h` donde Alan diseñó y documentó su TDA respectivamente.

Si Bárbara definiera su `main.c` como:

main.c

```c
#include <stdio.h>

#include "complejo.h"

int main(void) {
    complejo_t *a = complejo_crear_ri(1, 2);
    complejo_t *b = complejo_crear_ri(0, 1);

    complejo_t *r = complejo_sumar(a, b);

    printf("Real: %f, Imaginaria: %f\n", complejo_real(r),
        ↪ complejo_imaginaria(r));

    complejo_destruir(a);
    complejo_destruir(b);
    complejo_destruir(c);

    return 0;
}
```

Podríamos compilar este archivo como:

```
$ gcc -c main.c -Wall -std=c99 -pedantic
$
```

Obteniendo el objeto `main.o`.

Notar un detalle, la inclusión de `complejo.h` se hizó con comillas dobles (`"..."`) y no con *paréntesis angulares* (`<...>`) como hasta el momento. Cuando utilizamos los paréntesis el compilador sabe que tiene que ir a buscar el archivo de encabezados a la ruta por omisión donde

[1] `.so`: "shared object", otro tipo de objetos.

están los de sus bibliotecas, en cambio cuando utilizamos comillas estamos diciéndole que busque el encabezado como una ruta local relativa adonde está el código fuente que estamos compilando.

De un modo similar Alan incluirá a `complejo.h` en su fuente `complejo.c` para compilarlo. No mostramos el contenido completo del archivo `complejo.c` pero comenzará con:

`complejo.c`

```c
#include "complejo.h"

#include <stdlib.h>
#include <math.h>

struct complejo {
    float real, imaginaria;
};

// Y acá vendría la implementación de todas las primitivas
```

¿Para qué le sirve a Alan incluir su propio encabezado? Por un lado hay cosas que están definidas en el encabezado y no en el `.c`, como por ejemplo, la redefinición `typedef struct complejo complejo_t;`. Por el otro es una buena práctica porque forzamos a que las primitivas que Alan implemente en su código fuente coincidan con los prototipos que están en el archivo de encabezados.

Notar que por más que `complejo.h` sea "el" encabezado de Alan, Alan todavía tiene que incluir los encabezados que necesite para su código, `stdlib.h` para las funciones de memoria dinámica o `math.h` para hacer operaciones trigonométricas o de raíces. No sería correcto incluir esas inclusiones en `complejo.h` dado que no forman parte del contrato y es irrelevante para Bárbara saber qué utilizó internamente Alan para implementar el TDA.

En la próxima sección completaremos el archivo de encabezado.

14.3. Archivos de encabezados

Si bien ya en la sección 13.6 mostramos un esquema del archivo de encabezados profundizaremos un poco más acá.

Como ya se dijo dentro del archivo de encabezados habrán declaraciones de tipos, de funciones, etiquetas, etc.

Ahora bien, un archivo de encabezado tiene que estar diseñado de tal manera de que si uno lo incluyera en un archivo de fuentes el mismo no genere errores de compilación. Por ejemplo, si el encabezado contuviera una función `bool vector_asignar(const vector_t *v, size_t i, int elemento);`, ni `bool` ni `size_t` son cosas del lenguaje. Ya sabemos por experiencia que si no incluyéramos las bibliotecas donde ellos se definen el código no compilaría. Como no podemos asumir que quien incluya el encabezado incluya alguna cosa adicional es el mismo encabezado el que tiene que hacer los `#include` correspondientes.

Sabemos que `bool` está en `stdbool.h`, ahora bien, ¿dónde está `size_t`? Si vamos al caso, venimos utilizando este tipo desde hace rato y nunca mencionamos de dónde sale. No lo mencionamos porque si utilizamos `stdio.h` viene, pero también viene si utilizamos `stdlib.h`, pero también si utilizamos `string.h` y otros encabezados más. ¿Entonces está definido en todos? No, `size_t` está definida en un único encabezado, que es el mismo que define `NULL`, que se llama `stddef.h`. Todos los encabezados que mencionamos antes hacen un `#include <stddef.h>`.

Ahora bien si yo, por ejemplo, incluyera tanto `stdio.h` como `stdlib.h` estaría incluyendo dos veces a `stddef.h` y por lo tanto definiendo dos veces las cosas que están ahí. ¿Eso no sería un problema? Sí, de hecho sería un problema.

El preprocesador, además de para declarar etiquetas o incluir archivos sirve para realizar compilaciones condicionales. Esto es, activar o desactivar fragmentos de código según el estado de etiquetas[2].

Entonces, si tuviéramos un encabezado `miencabezado.h` envolveríamos el contenido del mismo en la siguiente construcción:

miencabezado.h

```
1 #ifndef MIENCABEZADO_H
2 #define MIENCABEZADO_H
3
4 // Acá estaría todo el contenido del encabezado
5
6 #endif
```

La instrucción **`#ifndef`** significa "si no está definido". Es decir, si no está definida la etiqueta `MIENCABEZADO_H` que, por qué lo estaría, si sólo debería haber un archivo con ese nombre, entonces hacemos dos cosas: En primer lugar la definimos (si no aclaramos nada, por omisión se define con el valor 1), para que esté para la próxima y en segundo lugar declaramos todas las cosas que queríamos declarar en el archivo de encabezado. El **`#endif`** es el indicador de que terminó el **`#ifndef`**, a diferencia de C, donde marcamos el final de los bloques con llaves en el preprocesador lo hacemos con esta instrucción.

En la primera inclusión del archivo se define todo lo que haga falta y además la etiqueta. Si hubiera una segunda inclusión no se entrará al **`#ifndef`** dado que `MIENCABEZADO_H` ya se encontraba definido de la vez anterior. Con esta salvaguarda se garantiza que las cosas se declaren una única vez. **Todos** los archivos de encabezado deben contener esta construcción, sin excepción.

Entonces, haciendo una puesta en común, el archivo `complejo.h` quedará:

complejo.h

```
1  #ifndef COMPLEJO_H
2  #define COMPLEJO_H
3
4  #include <stdbool.h>
5
6  typedef struct complejo complejo_t;
7
8  // Incluímos sólo algunas de las primitivas en este ejemplo:
9  complejo_t *complejo_crear_ri(float real, float imaginaria);
10 void complejo_destruir(complejo_t *c);
11 complejo_t *complejo_conjugar(const complejo_t *a);
12 complejo_t *complejo_sumar(const complejo_t *a, const complejo_t *
       ↪ b);
13 bool complejo_son_iguales(const complejo_t *a, const complejo_t *b
       ↪ );
14
15 #endif
```

Notar que la existencia de la función `bool complejo_son_iguales(...)` nos fuerza a incluir `stdbool.h`.

[2]Es lo que vimos cuando hablamos de `assert()`, que podía desactivarse compilando con `-DNDEBUG`, que no es otra cosa que decirle al compilador que defina una etiqueta `NDEBUG` de valor 1.

Y repetimos: No importa qué necesidades de encabezados requiera Alan para compilar `complejo.c`, en el encabezado del TDA sólo incluímos lo que es útil para el contrato y para que si Bárbara incluye el archivo su código compile.

14.4. Make

Si bien con las herramientas que vimos podemos compilar cualquier proyecto sin importar cuántos archivos posea primero compilando cada código fuente y luego enlazando todos juntos, ese proceso es tedioso e inconsistente. Presentaremos la herramienta Make que sirve para automatizar la compilación.

La herramienta Make utiliza un archivo de nombre `Makefile`[3] para definir las reglas de la compilación. Cada regla del archivo `Makefile` tiene el siguiente formato:

```
regla: dependencia1 dependencia2 ...
        accion1
        accion2
        ...
```

Importante el caracter que precede a las acciones es una tabulación (`'\t'`), Make no funciona si en vez de tabulación hay caracteres espacio, revisá cómo ingresar tabulaciones en tu editor de textos.

Antes de realizar la acción para la generación de la regla se chequearán las dependencias. Las dependencias a su vez pueden ser reglas o pueden ser archivos. Luego se ejecutarán la acciones. Por ejemplo:

```
complejo.o: complejo.c complejo.h
        gcc -c complejo.c -Wall -std=c99 -pedantic
```

Mi regla dice que si voy a construir `complejo.o` eso depende de los archivos `complejo.h` y `complejo.c` y que hay que ejecutar esa llamada al GCC.

Make no sólo es capaz de realizar la compilación, también Make es capaz de decidir si debe o no hacer la compilación. Si el archivo `complejo.o` no existiera obviamente debe compilarlo. Ahora bien, si el archivo ya existiera, ¿debe compilarlo de nuevo? Bueno, Make toma esa decisión en función de la fecha de modificación de la regla y de las dependencias. Si las fecha de modificación de las dependencias fuera posterior a la fecha de modificación de la regla eso indicaría que hubo cambios en mi código fuente y por lo tanto hay que recompilar. En cambio si la fecha de generación del objeto fuera posterior a la de los fuentes, eso significa que ya compilé la última versión y no necesito recompilar nada.

Esta optimización de qué compilar y qué no que hace Make hace que si estoy trabajando sobre un proyecto muy grande el mismo va a ser compilado completo sólo la primera vez. A partir de ahí sólo se recompilará lo que haga falta y luego se realizará el enlace final, donde el enlace es un proceso mucho más liviando que la compilación.

Antes de mostrar un ejemplo completo de `Makefile` observemos que si vamos a hacer muchas compilaciones vamos a tener que escribir muchas veces los parámetros del GCC. Ahora bien, tal vez queramos cambiar esos parámetros a futuro, por ejempo, queremos debuggear y agregar `-g` o queremos lanzar una versión en producción y agregar `-DNDEBUG`. En un caso así sería ineficiente tener que editar cada línea de compilación. Por suerte podemos definir etiquetas y utilizarlas después.

Dicho esto, el `Makefile` para compilar nuestro proyecto de complejos:

`Makefile`

[3] Sí, con eme mayúscula.

```
1  CFLAGS=-Wall -std=c99 -pedantic
2  LFLAGS=-lm
3
4  all: main
5
6  main: main.o complejo.o
7          gcc main.o complejo.o -o main $(LFLAGS)
8
9  main.o: main.c complejo.h
10         gcc $(CFLAGS) -c main.c
11
12 complejo.o: complejo.c complejo.h
13         gcc $(CFLAGS) -c complejo.c
14
15 clean:
16         rm *.o main
```

Expliquemos ahora el comando `make`. Podríamos ejecutar:

```
$ make
gcc -Wall -std=c99 -pedantic -c main.c
gcc -Wall -std=c99 -pedantic -c complejo.c
gcc main.o complejo.o -o main -lm
$
```

Make leyó la definición de dependencias del archivo `Makefile` y resolvió la primera regla `all`. Para construir `all` tuvo que construir `main` y eso disparó las diversas invocaciones al GCC.

Si modificáramos `main.c` e invocáramos de nuevo:

```
$ make
gcc -Wall -std=c99 -pedantic -c main.c
gcc main.o complejo.o -o main -lm
$
```

veríamos cómo no hubo necesidad de recompilar `complejo.o` porque las fechas indicaban que no había habido modificaciones en sus fuentes.

Notar que existe una regla `clean` que no es dependencia de ninguna, por lo tanto no va a ser invocada nunca. Podemos pedirle a Make que ejecute una regla puntual:

```
$ make clean
rm *.o main
$
```

En este caso borrará todos los archivos de la compilación y dejará el proyecto como antes de compilarlo. Esta regla suele incluirse tanto para limpiar el proyecto como para forzar el recompilado. Por ejemplo si modificáramos los flags de compilación del GCC en el archivo `Makefile` eso no modificaría ninguna de las dependencias pero querríamos volver a generar todo el proyecto. En un caso así habría que limpiar el proyecto primero.

Lo que se presentó en este capítulo es una introducción mínima a Make. Este programa es parte del ecosistema de aplicaciones de C y es un estándar. Cuando uno descarga bibliotecas o programas desarrollados en C espera que haya un archivo `Makefile` o similar[4]. Es un estándar descargar un proyecto y ejecutar `make all` para compilarlo.

[4]Existen alternativas superadoras a `make` para el caso de compilaciones complejas.

14.5. Entidades públicas y privadas

No se dijo de forma explícita, pero por lo que se vio hasta el momento si alguien define una función en un módulo `a.c` alguien puede utilizarla desde un módulo `b.c` tan sólo invocándola. Es cierto, hablamos de tener un `a.h` que declare a dicha función, pero incluso ante la ausencia de prototipo en el encabezado, la firma podría definirse en el archivo `b.c` y ser utilizada.

Dicho de otra forma, todas las funciones que definimos en nuestros archivos de código fuente están disponibles para ser utilizadas desde cualquier otro archivo de código fuente. Es decir, por omisión la visibilidad de las funciones es pública a los demás módulos.

En muchos casos podemos tener funciones que no tenga sentido que sean invocadas desde afuera. Por ejemplo si se trata de funciones auxiliares a un TDA, las mismas no formarían parte del contrato e incluso la existencia de esas funciones podría exponer cómo es la representación interna.

Podemos hacer que las funciones sean privadas a un único módulo anteponiendo la palabra **`static`** a la definición de la misma. Por ejemplo, si en el archivo `a.c` definiéramos:

```
1 static int a() {
2     return 5;
3 }
```

La función `a()` podría ser invocada desde dentro de `a.c` pero sería invisible para el enlazador en el código objeto.

Es importante marcar como **`static`** las funciones que no formen parte de la interfaz no sólo por paranoia de que Bárbara las utilice si no porque una función no documentada puede colisionar con otra función de otro módulo si se definieran dos funciones con el mismo nombre. Si la función no es de utilidad hacia afuera debe quedar delimitada a su módulo.

Algo similar a las funciones pasa con las variables globales. Si definiéramos una variable global en un módulo la misma estaría visible para los demás y colisionaría si hubiera una variable global con el mismo nombre en otro módulo. Entonces podemos hacer:

```
1 static const float g = 9.81;
```

y la variable será privada del fuente donde se defina.

Ahora bien, ¿cómo hacemos si quisiéramos tener una variable global compartida entre diferentes módulos? Es decir, querríamos que una única variable global sea declarada en un deterimnado módulo pero usada desde otros. Si nos limitáramos a declarar la variable en dos módulos distintos tendríamos una colisión entre dos variables diferentes.

En este caso esto se resuelve "avisándole" al compilador de la existencia de una variable que está en otro módulo, donde el enlazador será el encargado de referenciar. Es algo similar a cuando declaramos una función con un prototipo.

Supongamos que queremos que una variable de `a.c` se exponga globalmente. En el archivo `a.c` simplemente definiremos la variable:

```
1 const float g = 9.81;
```

Notar que la variable es pública.

Luego en el archivo `a.h` le "avisaremos" al compilador que la variable existe:

```
1 extern const float g;
```

Con el modificador **`extern`** el compilador sabe que existe una variable llamada `g` constante de tipo flotante, pero no reserva memoria para ella en los lugares donde esté definida. Ahora bien, cuando compilemos `a.c` el compilador ahí tendrá la variable declarada y definida sin **`extern`** por lo que sí reservará memoria y además expondrá el símbolo en el `a.o`. Cuando el enlazador

combine los objetos tendrá múltiples módulos que requieren de `g` y sólo uno que lo defina, simplemente enlazará las referencias.

Por completitud mencionaremos otro significado contextual de **`static`** no relacionado con los anteriores. Cuando utilizamos **`static`** para una variable dentro de una función esa variable será una variable privada de dicha función pero en vez de vivir en la pila vivirá en el espacio de las variables globales. Es decir, será una variable de una función que tendrá persistencia entre distintas invocaciones a la función. Por ejemplo:

```
1 int contar() {
2     static int cuenta = 0;
3     return ++cuenta;
4 }
```

Cada vez que llamemos a la función nos devolverá un número más que la vez anterior.

Las variables **`static`** tienen varios usos, desde poder devolver punteros a memoria que persiste sin utilizar memoria dinámica (pero que se sobreescribe su contenido si llamo de nuevo a la función) a tener funciones que recuerdan cosas entre llamadas. Si querés ver ejemplos podés investigar sobre la función `asctime()` o sobre `strtok()`, para ver dos usos diferentes.

14.6. Macros de función

Si bien son una herramienta desrecomendada, por completitud vamos a hablar de las macros de función.

Se llaman macros de función a las macros del preprocesador que realizan reemplazos con parámetros. Por ejemplo:

```
1 #define DUPLICAR(x) x * 2
```

Cuando en el código escribamos por ejempo `DUPLICAR(3.14)` la macro se expandirá a `3.14 * 2`, que evaluará a 6.28.

Ahora bien es importante destacar que las macros no son funciones por lo tanto no hay una evaluación de expresiones para inicializar parámetros. Una macro como la anterior está mal escrita y nunca debe definirse así. ¿Qué pasaría si alguien escribiera `DUPLICAR(1 + 1)`? Eso se expandiría a `1 + 1 * 2` lo cual evaluaría a 3. Incluso podría pasar que en algún contexto de inclusión hasta el operador de mutiplicación se asociara con algo más y el resultado no fuera el correcto.

Al escribir macros de función tenemos que prevenir a toda costa que la precedencia cambie el orden de evaluación. Una macro escrita de forma correcta podría ser:

```
1 #define MAX(a, b) ((a) > (b) ? (a) : (b))
```

¿Se ve la cantidad de paréntesis? No puede haber menos que esos.

Supongamos la macro:

```
1 #define ES_MAYUSCULA(c) ((c) >= 'A' && (c) <= 'Z')
```

¿Qué pasaría si escribiéramos `ES_MAYUSCULA(getchar())`? Si expandiéramos la macro obtendríamos `((getchar())>= 'A'&& (getchar())<= 'Z')`... ¡Son dos llamadas a `getchar()`!

Este último problema no puede ser resuelto utilizando macros, se resuelve implementando funciones. Cuando invocamos una función la expresión que pasemos por parámetro se evalúa una única vezy ese valor se le pasa a la función. Además de que la función se evalúa como un todo, que hay conversiones claras de tipos, etc.

Es por esto que si bien las macros son una herramienta del lenguaje, deben ser utilizadas con criterio y conscientemente de que no son funciones.

Capítulo 15

Manejo de archivos

15.1. Introducción

En el mundo de la computación los archivos son las entidades en las que persistimos datos, es decir, queremos almacenar algo para recuperarlo después, esos datos deberán ser archivados de alguna forma.

Cuando hablamos de archivos hablamos de dos propiedades independientes entre sí, por un lado sus datos, es decir el contenido que queremos persistir y por el otro lado su metadata que es cómo se llama ese archivo, qué permisos tiene, en qué ubicación se encuentra.

Los archivos forman parte de un sistema de archivos. En un sistema de archivos los mismos se organizan en estructuras de archivos y directorios. Los directorios son entidades que dentro pueden contener archivos y directorios, es decir, una estructura recursiva. Los mismos sirven para organizar la información dado que podemos anidar nuestro archivo dentro de una secuencia de directorios que forme una cadena lógica, por ejemplo `Usuarios/Juan/TA130` para encontrar los archivos de la materia TA130 de Juan que es un Usuario en esa computadora.

En un mismo dispositivo pueden coexistir diferentes sistemas de archivos y además de distinto tipo. Los dispositivos de almacenamiento no dejan de ser otra cosa que memorias, como la RAM, pero que persisten los datos entre reconexiones. Siendo estos dispositivos memorias los mismos tienen posiciones numeradas y se utilizan para almacenar y recuperar los bytes en esas posiciones. Cuando hablamos de diferentes sistemas de archivos queremos decir que el formato en el cual se almacenan los datos y la metadata dentro del dispositivo pueden ser diferentes, habiendo muchos formatos distintos y coexistiendo más de uno en el mismo sistema operativo.

Incluso las tecnologías de los dispositivos pueden ser diferentes, por ejemplo, almacenando en discos magnéticos bits según su polaridad, o almacenando en dispositivos ópticos que pueden reflejar o no un bit, o almacenando en dispositivos de estado sólido que utilizan semiconductores que conducen o no.

El sistema de archivos es una capa de abstracción que nos provee el sistema operativo en el cual nosotros podemos pensar en estas entidades de archivos, con sus datos y su metadata, guardados dentro de directorios e independizándonos de dónde o cómo se almacenan esos archivos.

Incluso mientras que en sistemas Windows los archivos se identifican con su unidad (`C:`, `D:`, etc.) y cada dispositivo físico (o virtual) genera una unidad en el común de los sistemas operativos se omite el concepto de unidad y los dispositivos se pueden asociar a ubicaciones arbitrarias del árbol de directorios. Por ejemplo, en nuestro ejemplo anterior, tal vez la carpeta de usuario de Juan se almacena en un disco diferente al del resto de los usuarios con redundancia para que Juan no pierda los datos en caso de falla.

Más aún, dentro de la abstracción que consistuye el sistema de archivos, incluso pueden

mostrarse como archivos y directorios cosas que no son dispositivos físicos. Por ejemplo, unidades de red, o un dispositivo que enchufemos como un teléfono celular o una cámara de fotos, o cualquier entidad o protocolo que pueda pensarse como un canal donde escribir datos o del cual leer información.

La idea del sistema de archivos es uniformizar todos los detalles de implementación y simplificarlos en que un archivo tiene una ruta, que será su ubicación dentro del árbol de unidades y directorios y que si conozco esa ruta (y tengo los permisos suficientes) puedo acceder a su contenido. Si eso realmente es un archivo o es otro tipo de entidad virtual o si está almacenado remotamente no nos interesa.

15.2. Interacción con los archivos

Dado que el concepto de archivo es una abstracción y que constituye una interfaz de alto nivel que provee el sistema operativo, al igual que en el caso de la memoria será un recurso que gestionaremos a través de él.

De forma simplificada la secuencia empezará diciéndole al sistema operativo que queremos tener acceso a determinado archivo. Para referirnos a nuestro archivo lo haremos por su ruta, donde llamamos ruta a la secuencia de unidades y directorios que dan la ubicación única del archivo. Por ejemplo:

```
/home/juan/TA130/ej1.c
```

podría ser la ruta completa del `ej1.c` que el usuario Juan tiene en su carpeta personal en un sistema basado en Unix, mientras que

```
D:\Usuarios\Juan\TA130\ej1.c
```

podría ser una ruta similar en un sistema Windows. En el caso de Windows `D:` será una unidad, por ejemplo una partición determinada de un disco rígido mientras que en el caso de Unix todos los archivos penden del directorio raíz `/` sin importar su unidad, las unidades se "montan" en cualquier ruta. También cambian en uno y otro caso los separadores de directorio siendo `/` en Unix y `\` en Windows. Y hay más diferencias, como por ejemplo que en Unix importan las mayúsculas y minúsculas, no existe el concepto de extensión, los nombres pueden contener cualquier carácter y una serie más de características en las que no vamos a profundizar.

Las rutas que acabamos de dar, que comienzan con `/` en el caso de Unix y con la unidad (por ejemplo `D:`) en el caso de Windows se denominan rutas absolutas. Es decir, hay un solo archivo que responde a esa ruta en todo el sistema operativo y eso constituye un identificador único. No importa en dónde esté ubicado, la ruta absoluta me permite identificar un archivo de forma unívoca.

Ahora bien, en ambos sistemas, si estuviera ubicado en la respectiva carpeta personal de Juan y quisiera referirme al mismo archivo podría hacerlo como `TA130/ej1.c` (o con las barras invertidas en Windows). Notar que esta ruta no empieza ni con la raíz ni con la unidad, es una ruta relativa. Si estoy parado en `/home/juan` esta ruta se traducirá como `/home/juan/TA130/ej1.c`, pero si estuviera en `/usr/lib` se convertirá en `/usr/lib/TA130/ej1.c`. Se dice entonces que es una ruta relativa, porque según dónde esté posicionado va a hacer referencia a diferentes ubicaciones.

Tiene sentido utilizar rutas relativas por simplicidad porque son más cortas, pero también cuando desconocemos la ruta absoluta. Por ejemplo, cuando hablamos de modularización en el capítulo 14 nosotros podemos distribuir un proyecto para que cualquier persona se lo descargue y lo compile en su computadora. Nosotros no sabemos en qué lugar de la computadora va a descargarlo, ni tampoco podemos imponerlo. Entonces, cuando incluímos un archivo haciendo `#include "archivo.h"` utilizamos rutas relativas. Es decir, ese archivo `.h` deberá estar en la

misma ruta que el `.c` que lo incluye. Eso le da flexibilidad al usuario de compilarlo donde prefiera.

Volviendo a la interacción con el sistema operativo, entonces nosotros le pediremos abrir el archivo dándole su ruta. Esta operación podrá concretarse o no dependiendo de si la ruta es correcta, el archivo existe, los permisos de mi usuario son suficientes para manipularlo, el archivo no está ocupado y un sinfín de eventualidades que nos exceden y podrían suceder. A diferencia de los pedidos de memoria donde una falla en el pedido indicaría que se agotó la memoria total del dispositivo y siempre podría decir "voy a pedir sólo 2 bytes, no puede fallar"[1] hay tantas cosas externas que condicionan el acceso a un archivo que **nunca** podemos asumir que pudo abrirse sin chequearlo. Asumamos que el archivo pudo abrirse.

Los archivos son entidades de acceso secuencial, lo que ya llamamos streams (o flujos). Cuando abrimos un archivo la abstracción setea un cursor al primer byte de dicho archivo. Cada vez que hagamos, por ejemplo, una operación de lectura, se devolverá el valor en ese byte y se adelantará una posición el cursor. Si hiciésemos una operación de escritura, se escribirá en ese byte y se adelantará el cursor. Es decir, los archivos se recorren desde el comienzo hasta el final avanzando de a una posición por vez de forma automática.

Si bien existe la posibilidad manipular el cursor para acceder a posiciones de forma aleatoria, preferiremos no hacerlo nunca. Todos los dispositivos físicos están diseñados para acceder en forma secuencial. Incluso hay dispositivos que no soportan ni rebobinar ni avanzar en el tiempo. Esto es parte del concepto de stream, tenemos que considerar a la información como una secuencia de bytes que recibimos o enviamos con un determinado orden, si los dejamos pasar los perdemos, si ya los emitimos no podemos arrepentirnos. Es el concepto de entrada salida que vimos cuando hablamos de la interacción con el usuario. ¿Qué sería retroceder la entrada del usuario?, ¿pedirle que vuelva a ingresar lo que ya ingresó? Entonces en el modelo de archivos de C vamos a considerar que toda lectura o escritura se realiza de forma secuencial y en una única pasada.

Una vez abierto el archivo, con el cursor puesto en la primera posición podremos escribir o leer tantas veces como queramos o hasta agotar el recurso. Cuando terminemos esta operación entonces tendremos que liberarlo. Al igual que lo que dijimos con respecto a la apertura, la liberación es mucho más importante que la de la memoria, no devolver un archivo podría implicar que los cambios que realicemos en él nunca se vuelquen físicamente en el dispositivo, podría dejarlo inaccesible en el sistema de archivos, podría incluso corromperlo. Los archivos son recursos escasos y además recursos compartidos, tenemos que minimizar el tiempo que nuestras aplicaciones los bloquean.

15.3. El tipo `FILE`

Siendo que los archivos son una abstracción del sistema operativo no es de extrañar que la implementación de archivos en la biblioteca de C sea a través de un TDA. Como parte de la funcionalidad de entrada/salida el compilador implementa el tipo `FILE` que sirve para manipular archivos. Siendo un TDA tendrá un constructor, que se corresponde con abrir el archivo, un destructor, que lo cierra y libera sus recursos, y luego primitivas que sirven para leer o para escribir datos en él.

Empecemos por ellos:

```
1 FILE *fopen(const char *ruta, const char *modo);
2 int fclose(FILE *f);
```

El constructor `fopen()` intenta abrir el archivo dado por la `ruta` y en el `modo` que le indiquemos y nos devuelve el TDA creado o `NULL` en caso de falla. Cabe destacar que en la ruta

[1] No, tampoco pueden hacer esa asunción en el curso.

podemos utilizar / como separador de directorios independientemente de la plataforma en la que estemos, esto es importante porque las barras invertidas en C son el carácter de escape y es muy frecuente olvidarse de *escapear* una barra. El destructor `fclose()` libera los recursos del archivo `f`, devuelve `0` si el archivo no tuvo problemas o `EOF` si sí los tuvo. Cuando se habla de "tener problemas" no es durante el cierre sino sobre toda la vida del archivo. Si bien todas las primitivas del archivo nos indican si hubo éxito o no en la operación que quisimos realizar, muchas veces es engorroso validar en cada una de las operaciones de lectura o de escritura. Si hubiera una falla la misma sería recordada por el TDA y al momendo del cierre, si no es que es ya muy tarde, podemos validar una única vez si todo el proceso fue exitoso o no. Dependiendo de la aplicación puede ser adecuado o no.

15.3.1. El modo

El segundo parámetro del constructor es el modo, el modo lo que va a dar es información de qué queremos hacer con el archivo. El estándar provee múltiples modos pero nosotros nos centraremos en solamente 3, dado que los demás no son usuales en programación de alto nivel.

A diferencia de lo que se explicó genéricamente, en C se suelen abrir flujos o de lectura o de escritura, pero no de ambas a la vez. El modo principalmente indicará si estamos abriendo el archivo para leer de él o para escribir en él. Si el archivo fue abierto para lectura tendremos que utilizar primitivas de lectura, si lo abrimos para escritura tendremos que utilizar primitivas de escritura.

Los modos más usuales son:

`r`: Abre el archivo en modo lectura (*read*). Si el archivo no existe falla.

`w`: Abre el archivo en modo escritura (*write*). Si el archivo no existe lo crea, si el archivo existe lo trunca, esto es elimina **todo** su contenido.

`a`: Abre el archivo en modo añadidura (*append*), que es un modo de escritura. Si el archivo no existe lo crea, si el archivo existe el cursor se ubica al final del mismo, es decir lo que escriba se escribirá al final de lo que ya estaba.

Cabe aclarar que en todos los casos también hay fallas si las rutas son incorrectas, los permisos no son adecuados, etc.

Hay otra indicación de modo que es ortogonal a estas vistas que retomaremos más adelante.

15.4. Archivos de texto

Llamamos archivos de texto, o de texto sencillo (o texto plano por una mala traducción de *plain text*) a los archivos que contienen caracteres. Es decir, archivos que están pensados para ser legibles por un ser humano.

Cuando manipulemos archivos de texto tendremos, al igual que cuando hablamos de interacción con el usuario (ver sección 6.8), dos estrategias principales: Interactuar de a un carácter por vez o interactuar de a líneas.

Las funciones de lectura en archivos de texto son:

```
1 int fgetc(FILE *f);
2 char *fgets(char *s, int size, FILE *f);
```

La función `fgetc()` lee un carácter de un archivo `f` (abierto en modo lectura, no habría que aclararlo) y lo devuevle, en caso de falla devuelve `EOF`. La función `fgets()` es una vieja conocida, lee una línea del archivo `f` hasta alcanzar el `'\n'` o `size`-1 caracteres y la almacena en `s`, devuelve `s` o `NULL` en caso de falla.

En ambos casos cuando decimos falla podemos querer decir tanto que algo falló realmente o que se terminó el archivo `f` y ya no hay nada más que leer. En C no podemos anticipar cuándo se termina un archivo, nos enteraremos que se terminó cuando intentemos leer más allá del final. La lectura del último carácter será totalmente normal y el archivo todavía estará no terminado incluso aunque el cursor ya se haya incrementado más allá del final. Al leer con el cursor más allá del final se disparará la señal de final de archivo.

De modo análogo, las funciones de escritura de archivos de texto son:

```
1 int fputc(int c, FILE *f);
2 int fputs(const char *s, FILE *f);
3 int fprintf(FILE *f, const char *formato, ...);
```

La función `fputc()` escribe el carácter `c` en el archivo `f`. La función `fputs()` imprime la cadena `s` en el archivo `f`. La función `fprintf()` imprime según el `formato` en el archivo `f`. Todas estas funciones devuelven un número positivo si todo funcionó bien (en el caso de `fputc()` el carácter impreso, en las otras el número de bytes que escribieron) o `EOF` en caso de falla.

Por ejemplo:

```
1  #include <stdio.h>
2
3  int main(void) {
4      FILE *f = fopen("TA130/ej1.c", "r");
5      if(f == NULL) {
6          fprintf(stderr, "No␣pudo␣abrirse␣el␣archivo.\n");
7          return 1;
8      }
9
10     int c;
11     while((c = fgetc(f)) != EOF)
12         putchar(c);
13
14     fclose(f);
15     return 0;
16 }
```

Abre el archivo de ruta relativa `TA130/ej1.c`[2] en modo lectura, luego lee del mismo de a un carácter por vez hasta que se termine la entrada e imprime cada uno de esos caracteres por `stdout`.

15.4.1. `stdin`, `stdout` y `stderr`

Señalemos el elefante en la habitación, venimos utilizando varias de las funciones de archivos desde que hablamos de interacción con el usuario en la sección 6.8, y esto es porque los flujos de C internamente son archivos. Es decir, en algún lugar está definido:

```
1 FILE *stdin, *stout, *stderr;
```

y es el compilador el que se encarga de abrir estos archivos si los utilizamos.[3]

Siendo que ya estamos familarizados con flujos de texto, e incluso con utilizar archivos como si fueran flujos (ver 6.8.5) utilicemos ese conocimiento previo para asimilar archivos.

[2] O `TA130\ej1.c`, si estuviéramos en Windows.

[3] Si en algún momento notaste que Valgrind te indicaba pedidos de memoria que no hiciste, probablemente sean los buffers de estos flujos.

Como se viene diciendo repetidas veces en este capítulo interactuar con archivos es similar a interactuar con el usuario, hay caracteres en secuencia que podemos leer o escribir. Las funciones para leerlos y escribirlos en algunos casos son similares, en otros casos son literalmente la misma función. En muchos casos las funciones de interacción son wrappers de las funciones de archivos, por ejemplo `putchar(c);` no es otra cosa que `fputc(c, stdout);`, `getchar();` es `fgetc(stdin);` y `printf(...);` es `fprintf(stdout, ...);`.

Y **¡atención!** la biblioteca es inconsistente, `puts(s);` y `fputs(s, f);` no son equivalentes. La primera imprime por `stdout` la cadena `s` y **además** imprime un `'\n'`. En cambio `fputs()` imprime solamente `s` sin agregar nada más.

15.4.2. El '\n'

Antes que nada reiteremos lo que dijimos en la sección anterior, los flujos con los que estamos interactuando desde el hola mundo son archivos, conocemos cómo se comportan. Recordemos entonces el hola mundo:

```
1 printf("Hola␣mundo\n");
```

En su momento dijimos que el `'\n'`, el caracter de *line feed* (LF), imprimía "un enter" al final de la línea. Y está bien la afirmación, pero es más complejo.

El carácter `'\n'` es el carácter número 10 en la tabla ASCII. Ahora bien, sólo en sistemas operativos derivados de Unix `'\n'` dispara un final de línea.

En sistemas operativos derivados de Macintosh (Apple) el final de línea se dispara con el carácter `'\r'`, el carácter de *carriage return* (CR), el 13 de la tabla ASCII.

Y en sistemas operativos derivados de DOS (Microsoft) el final de línea se dispara con la secuencia `"\r\n"` (CR-LF).

Con esto queremos decir que diferentes terminal necesitan diferentes secuencias de caracteres para generar el final de línea. Ahora bien, cuando presentamos el hola mundo no hicimos ningún comentario al respecto de la portabilidad de haber finalizado con `'\n'` en distintas plataformas... y no lo hicimos porque el hola mundo que presentamos **ya** es portable.[4]

El compilador conoce la plataforma en la cual estamos y sabe que semánticamente para C imprimir un `'\n'` significa "imprimir un final de línea". Entonces, se toma la libertad de reemplazar tanto en las operaciones de entrada como en las operaciones de salida esta entidad por lo que corresponda. Esto quiere decir que cuando manipulemos streams de texto en Windows escribir un `'\n'` se reemplazará por escribir la secuencia `{13, 10}` y también significa que si estamos en un sistema operativo de Apple cuando se lea el byte 13 a nosotros nos llegará el carácter `'\n'`. Es decir, se hará una traducción en uno y otro sentido para que para nosotros sea natural que `'\n'` es el finalizador de líneas, como en Unix, el sistema operativo que le dió nacimiento al lenguaje C.[5]

Esto que estamos mencionando es una característica que nos interesa particularmente para archivos de texto, y profundizaremos en esto en la siguiente sección.

Siendo que cuando compilemos un programa para Windows funcionará en Windows y cuando compilemos un programa para Mac funcionará en Mac los archivos de texto[6] serán consistentes con la plataforma en la que estemos. La incompatibilidad de los archivos de textos en diferentes plataformas nos interesará sólo si quisiéramos abrir un archivo generado en Windows en un sistema operativo con otra convención de finalización de línea, pero esto será algo que no podremos resolver con archivos de texto.

[4] Voy a omitir decir esta vez *"y con esto terminamos de entender el hola mundo"* porque ya debe ser la décima vez que lo decimos en este apunte.

[5] Similar a lo que dijimos del separador de directorios en las rutas para `fopen()`.

[6] Y los flujos estándar de la terminal.

15.5. Archivos binarios

En contraposición con los archivos de texto, los archivos binarios no están pensados para ser legibles, en un archivo binario se vuelcan variables del mismo modo (o de forma similar) a como están en memoria. Es decir, si en la memoria tengo un entero de 16 bits de contenido `0xDEAD` en un archivo de texto podría imprimirse por ejemplo como `"57005\n"` una secuencia de caracteres en base decimal. En binario sencillamente se podría escribir como la secuencia de bytes `{0xDE, 0xAD}`... aunque también podría escribirse como la secuencia `{0xAD, 0xDE}`. Antes de explicar por qué hay dos formas posibles de escritura cerremos la idea, y digamos, se escribirán 16 bits, tal cual como esos bits estaban en la memoria. Son 2 bytes, nada más, no hay una traducción a ASCII. Y escribir los datos tal cual como estaban en la memoria implica que cuando se lean se pueden subir a la memoria tal cual como están en el archivo y eso ya representará de nuevo la cantidad `0xDEAD`.

Al respecto de las dos formas de almacenar `0xDEAD` en memoria hay platarformas donde los bytes de los números se almacenan primero los más pesados y después los más livianos y otras en las que se almacenan al revés. La primera convención se llama *big-endian* mientras que la otra se llama *little-endian*. Como con todo, si hasta ahora no nos preocupamos por esto, ni siquiera cuando presentamos el manejo de bits a bajo nivel es porque esto es un detalle de implementación que no afecta a ninguna operación. Es más ni siquiera es sencillo diagnosticar si estamos en una plataforma de un tipo o del otro... ahora bien, esto sí nos interesará si quisiéramos levantar en una plataforma lo que generamos en otra. Aunque también nos preocupará si los enteros tienen el mismo tamaño en una u otra, o cómo se alinean las estructuras, etc.

Se puede decir que hay dos universos totalmente disjuntos de aplicaciones de archivos binarios: El primero es persistir entidades de la memoria de mi aplicación en un achivo para recuperarlas luego. A diferencia de escribir en modo texto, donde tengo que procesar los datos y convertirlos, escribir datos binarios se limita sencillamente a realizar un volcado de la memoria. Por lo que dijimos en el párrafo anterior, esta persistencia depende de que nunca pretenda llevarme ese volcado a otra plataforma. El segundo es codificar dentro de un archivo datos de algún formato bien conocido, como formatos de imágenes, de documentos, de audio, video, etc. En este caso lo importante es la interoperabilidad, no importa en qué dispositivo abramos una imagen JPEG, queremos ver la imagen que ella representa, es por eso que en ese tipo de formatos habrán especificaciones formales que dirán cómo se codifican los datos, con qué tamaños, con qué endianness, etc.

15.5.1. El modo binario

Imaginemos que tenemos el entero de 16 bits `0x0D0A`, prestar atención a sus bytes que no son otros que `{13, 10}`[7] o, expresado de otra forma, `{'\r', '\n'}`. Si quisiéramos escribir esa secuencia de bytes en un sistema Windows dijimos que al querer escribir `'\n'` la capa de abstracción del compilador lo reemplazará por `"\r\n"`. Análogamente, si tuviéramos un archivo que contuviera la secuencia `0x0D0A` y lo leyéramos en Windows obtendríamos en vez de dos bytes apenas un `'\n'`. Vamos a tener estos problemas en todas las plataformas salvo en Unix donde lo que representamos en C como `'\n'` se representa en el sistema operativo del mismo modo.

Es evidente que esta característica que nos resulta tan cómoda para manipular archivos de texto va a romper por completo el procesamiento de archivos binarios. Por eso es que hay un modo adicional a los que ya vimos que es el modo `b` (*binary*) que se puede agregar a los modos que ya vimos, por ejemplo `"wb"` para escribir en modo binario. Abrir un archivo en modo binario desactiva la traducción del `'\n'`, en modo binario al archivo le llega exactamente

[7] En el orden que corresponda según el endianness de la plataforma.

lo que se escribió y lo mismo vale para la lectura.

15.5.2. Funciones

Abrir un archivo en modo binario no inhabilita usar las primitivas que ya vimos de lectura y escritura, pero probablemente no nos sirvan para las operaciones que queremos realizar.

Como dijimos, la manipulación binaria implica querer volcar desde y hacia la memoria, y se provee un par de funciones para realizar esta operación:

```
1 size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *f)
    ↪ ;
2 size_t fread(void *ptr, size_t size, size_t nmemb, FILE *f);
```

Como se ve ambas funciones tienen prácticamene la misma firma. La primera escribe en un archivo (es decir vuelca desde la memoria), la segunda lee de un archivo (es decir vuelca hacia la memoria). Los datos en memoria están en `ptr`, cada uno de los datos mide `size`, hay `nmemb` datos y el archivo a operar es `f`. Las funciones devuelven cuántos datos pudieron escribir/leer respectivamente.

Por ejemplo, si quisiéramos volcar un vector de enteros en un archivo:

```
1  #include <stdio.h>
2
3  int main(void) {
4      int v[5] = {1, 2, 3, 4, 5};
5
6      FILE *f = fopen("vector.bin", "wb");
7      if(f == NULL) {
8          fprintf(stderr, "No␣pudo␣abrirse␣el␣archivo\n");
9          return 1;
10     }
11
12     if(fwrite(v, sizeof(int), 5, f) != 5) {
13         fprintf(stderr, "Falló␣la␣escritura␣de␣alguno␣de␣los␣datos
               ↪ \n");
14         fclose(f);
15         return 1;
16     }
17
18     if(fclose(f) == EOF) {
19         fprintf(stderr, "Hubo␣alguna␣falla␣escribiendo␣en␣el␣
               ↪ archivo\n");
20         return 1;
21     }
22
23     return 0;
24 }
```

Abrimos un archivo relativo `vector.bin` para escribir en modo binario. Intentamos escribir los 5 enteros del vector `v` en bloque. Como la función nos indica cuántos elementos pudo escribir debería devolvernos que escribió los 5, cualquier otro valor que devuelva será un error. Luego estamos también validando el valor de retorno de `fclose()`. Recordemos que dijimos que al cerrar el archivo nos avisaba si había habido algún error a lo largo de toda la manipulación del archivo, por lo que en este ejemplo estamos validando de forma redundante. Por lo general,

según la criticidad del problema, nos interesará o validar cada una de las lecturas/escrituras o sencillamente validar únicamente al cerrarlo.

Miremos el contenido del archivo en una plataforma x86:

```
$ hd vector.bin
00000000  01 00 00 00 02 00 00 00  03 00 00 00 04 00 00 00  |................|
00000010  05 00 00 00                                       |....|
00000014
$
```

El comando `hd` (*hex dump*) muestra cada uno de los bytes de un archivo en formato hexadecimal. La primera columa es el índice, luego vienen los bytes, y la última columna mostrará la representación ASCII en caso de tratarse de caracteres imprimibles (en este caso ninguno lo es). Como se puede observar x86 es una plataforma de 32 bits de tipo little-endian.

El parámetro de retorno de `fread()` no sólo nos sirve para validar lo que intentamos leer, también nos puede servir para leer los datos que haya en un archivo, por ejemplo con el siguiente fragmento:

```
1 FILE *f = fopen("vector.bin", "rb");
2 if(f == NULL) return 1;
3 int v[100];
4 size_t n = fread(v, sizeof(int), 100, f);
5 // n == 5
6 fclose(f);
```

Intentamos leer hasta 100 enteros del archivo que generamos antes, como el archivo se termina al quinto entero nos dirá que pudo leer sólo 5, no importa que no hayamos leído 100. Es más, si hubierámos leído 100 daría para sospechar si quedaron enteros sin leer del archivo.

15.5.3. Lectura independiente del endianness

Los ejemplos anteriores son volcados de memoria en crudo y dependen de la plataforma. Dijimos que esa era una de las aplicaciones de archivos binarios, pero también dijimos que hay todo un universo donde querremos leer o escribir formatos conocidos donde se especifica un determinado endianness.

Si tuviera que escribir un entero de 32 bits en little-endian y supiera que estoy en x86 y con un GCC que escribe números de 32 bits, podría hacer lo que hice antes. Ahora bien, mi código no va a ser portable. Si alguien compilara el mismo código en otra plataforma no estaría garantizado ni que se escribieran 32 bits ni el endianness. Entonces veamos cómo podemos hacer para leer o escribir en un formato específico independientemente de la plataforma en la que estemos.

Supongamos que sabemos que un determinado formato contiene un entero de 32 bits escrito en little-endian. Dado que el endianness afecta sólo a datos multibyte si nosotros leyéramos

```
1 uint8_t bytes[4];
2 fread(bytes, 1, 4, f);
```

sabríamos que en `bytes[0]` está el byte más liviano, en `bytes[1]` el siguiente y así, porque nos dijeron que el archivo codificaba enteros de 32 bits en little-endian.

¿Sabemos formar un número de 32 bits juntando 4 bytes separados? Claro que sabemos, lo vimos en el capítulo 10:

```
1 uint32_t dato = bytes[0] | bytes[1] << 8 | bytes[2] << 16 | bytes
    ↪ [3] << 24;
```

No olvidemos de que el endianness es algo interno de la arquitectura que no afecta a cómo vemos las operaciones, el byte que desplazamos 24 veces a la izquierda va a quedar en el byte más pesado de `dato`. No importa si en la memoria ese byte se almacena en `&dato` o 3 bytes más adelante, será el byte más pesado.

Análogamente podemos descomponer cualquier dato de cualquier tamaño en los bytes que lo componen y ordenarlos para escribirlos de forma individual en el endianness que queramos, sin importar el endianness de nuestra plataforma.

Capítulo 16

Argumentos en Línea de Comandos (CLA)

16.1. Argumentos

Recordemos la compilación del hola mundo:

```
$ gcc hola.c -o hola.exe -std=c99
```

El GCC es un programa, particularmente programado en C[1], y está obteniendo información del usuario (de nosotros) de una forma diferente a la que nosotros conocemos. El GCC no es un programa interactivo, es decir, nunca nos pregunta cosas que nosotros ingresamos por `stdin` si no que nosotros le decimos qué queremos hacer en el momento de la ejecución pasándole parámetros en la invocación.

El estándar de C admite dos posibles firmas para la función `main()`. Una es la firma que ya vimos `int main(void)` mientras que la otra firma es: `nt man`(int argc, char *argv[]) ...

Esta firma recibe dos parámetros: Un entero y un arreglo de cadenas de caracteres. Si bien es convención llamar a estos parámetros `argc` (cantidad de argumentos) y `argv` (vector de argumentos) respectivamente, podríamos usar cualquier nombre[2]. El sistema operativo al realizar la invocación será el responsable por pasarle estos parámetros al `main()` ya que esta función es el punto de entrada.

En el ejemplo del GCC que vimos al comienzo el sistema operativo generaría esta llamada: `main(5, {"gcc", "hola.c", "-o", "hola.exe", "-std=c99", NULL});`. Es decir, como su nombre lo indica, `argc` es la cantidad de argumentos suministrados y `argv` es un vector con cada uno de ellos. Notar que `argv` mide uno más, porque contiene el centinela `NULL` para indicar la finalización, lo cual es redundante con `argc` dado que `argv[argc] == NULL`. Notar que el primer argumento es el nombre del programa, entonces nunca puede haber menos de un argumento porque el programa fue invocado para ejecutarlo.

Sobre el tema en sí no hay mucho más que decir, lo complejo no es qué recibe un programa si no cómo operarlo después. Es decir, ¿cómo hace el GCC para activar o desactivar opciones en función de sus argumentos, procesar estos argumentos, etc? Bueno, eso ya tendrá que ver con la funcionalidad del GCC.

Los argumentos en línea de comandos son útiles cuando queremos generar programas no interactivos, es decir, que no requieran un operador ejecutándolos y su invocación está automatizada. También son prácticos cuando tenemos programas que tienen muchas opciones y modos de funcionamiento[3], es mucho más práctico que el usuario lea el manual y evalúe

[1] ¿Con qué compilador de C compilamos un compilador de C programado en C?, ¿eh?
[2] Pero es convención llamarlos así, entonces no uses otro nombre.
[3] Si bien nosotros sólo vimos una decena, el GCC tiene más de 5000 parámetros.

cuáles opciones quiere activar y las proporcione al invocar a que el programa le pregunte por cada una de esas opciones.

Cabe destacar que recibir opciones por argumentos no es excluyente con interactuar luego con el usuario, son dos maneras de controlar el funcionamiento de un programa.

16.2. Uso de argumentos

Veamos un ejemplo de un programa con argumentos:

sumar.c

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    if(argc != 3) {
        fprintf(stderr, "Suma␣dos␣números.␣Uso:␣%s␣<num1>␣<num2>\n
            ↪ ", argv[0]);
        return 1;
    }

    float n1 = atof(argv[1]);
    float n2 = atof(argv[2]);

    printf("%f\n", n1 + n2);

    return 0;
}
```

Se espera que el programa se use así:

```
$ ./sumar
Suma dos números. Uso: ./sumar <num1> <num2>
$ ./sumar 4 7
11
$
```

Siendo que el programa no interactúa con el usuario, entonces el programa tiene que poder guiar a un usuario que no sepa cómo invocar al programa. Dada la invocación que nosotros esperamos donde necesitamos recibir dos números además del nombre del programa, entonces será válida una invocación con 3 argumentos. Ese es el chequeo que hacemos para mostrar la ayuda si no se valida. Notar como al mostrar la ayuda podemos obtener el nombre de nuestro programa que está contenido en `argv[0]`.

Más allá de educar al usuario en el uso del programa, notar que más adelante queremos acceder al contenido de `argv[1]` y `argv[2]`. Si no validamos primero la existencia de esos argumentos (sea mirando `argc` o iterando `argv` hasta el centinela) no podemos acceder a ellos con seguridad. Si el programa hubiera sido invocado sin argumentos en `argv[1]` tendríamos un `NULL` y acceder a `argv[2]` sería directamente una violación de memoria. No puede accederse a argumentos sin validar antes que existan.

Luego hay que tomar en cuenta que, similar a la interacción con el usuario, el usuario ingresa cadenas de caracteres. Si necesitáramos otros tipos deberemos hacer las conversiones correspondientes. Además, al tratarse de entrada del usuario habrá que validar que los valores sean válidos[4].

[4]En nuestro ejemplo convertimos con `atof()`, pero si quisiéramos verificar que realmente se tratara de números

16.3. Comodines

Te recomendamos que juegues con este ejemplo:

argumentos.c

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("argc = %d\n", argc);
    for(size_t i = 0; argv[i] != NULL; i++)
        printf("argv[%zd] = \"%s\"\n", i, argv[i]);
    return 1;
}
```

¿Qué salida ves si ejecutás?

```
$ ./argumentos hola que tal
$ ./argumentos hola    que    tal
$ ./argumentos hola\ que tal
$ ./argumentos hola "que  tal"
$ ./argumentos hola que tal *
$ ./argumentos hola que tal *.*
```

No vamos a explicar los primeros, fijate, pero vamos a profundizar en el `*`.

Como dijimos los argumentos los genera el sistema operativo, y no necesariamente los genera de forma literal de lo que el usuario escribe. Algunos de ellos constituyen expresiones.

En la sección 14.4 cuando vimos la confección del archivo `Makefile` en la etiqueta `clean` escribimos `rm *.o` y dijimos que eso borraba todos los códigos objeto. Los código objeto terminan en `.o` y el asterisco es un comodín que significa "cualquier cosa", es decir, el patrón `*.o` significa "los nombres de los archivos que tengan cualquier cosa y terminen en .o". El sistema operativo confecciona una lista de los archivos que validen ese patrón y se los pasa como argumentos al programa `rm` en este caso.

Similar al asterisco está el comodín `?`, el mismo significa "cualquier letra". Por ejemplo el patrón `archivo?.c` va a validar contra los archivos `archivo1.c` y `archivos.c` pero no contra `archivo42.c`.

Si bien todas las terminales implementan estos comodines, puede haber diferencias en cómo funcionan en Windows dado que en ese sistema operativo los nombres de los archivos contienen extensiones y eso cambia su comportamiento. Además en terminales particulares hay otros patrones mucho más complejos que estos dos presentados acá.

podríamos haber utilizado `strtof()`.

Capítulo 17

Complejidad Computacional

17.1. Eficiencia

En la sección 11.7 se presentó un ejemplo muy similar a este:

```c
/* Lee enteros de stdin hasta agotar la entrada, devuelve el
vector de enteros leídos por el nombre y la cantidad de enteros
a través de n. */
int *leer_enteros(size_t *n) {
    int *v = NULL;
    size_t i = 0;

    char buffer[100];
    while(fgets(buffer, 100, stdin) != NULL) {
        int *aux = realloc(v, (i + 1) * sizeof(int));
        if(aux == NULL) {
            free(v);
            return NULL;
        }
        v = aux;
        v[i++] = atoi(buffer);
    }

    *n = i;
    return v;
}
```

en el cual se lee de la entrada una cantidad desconocida n de enteros. La pregunta que queremos hacernos es si la eficiencia de nuestra implementación depende de n y en qué medida lo hace.

Antes que nada, cuando hablamos de eficiencia hablamos principalmente de dos cantidades: Tiempo y recursos. El tiempo tendrá que ver con la cantidad de operaciones que se realicen y los recursos que consuma será la cantidad de memoria requerida. Profundizaremos más sobre esto más adelante, pero en principio no nos interesa tanto el tiempo o la cantidad de bytes de memoria[1] si no cuál es su relación con el tamaño n del problema. Es decir, si se duplica n, ¿qué pasará con los recursos?, ¿serán los mismos?, ¿se duplicarán?, ¿se multiplicarán por, por ejemplo, 4?[2]

[1] Donde la cantidad de memoria podemos calcularla, pero el tiempo dependerá de dónde vayamos a correr esto.
[2] Hagan sus apuestas.

Para simplificar diremos que asignar una variable, obtener el valor de ella, hacer una operación aritmética, etc. son operaciones que llevan siempre el mismo tiempo. O sea, no dependen de qué tan grande sea el número que asigno, o si estoy operando dos números muy grandes, etc.

Si miramos el código vamos a ver que hay una secuencia de estas operaciones sencillas, seguidas de un ciclo que se va a ejecutar n veces. Dentro de este ciclo tenemos operaciones sencillas con excepción de una llamada a `fgets()`, una llamada a `realloc()` y otra a `atoi()`. Si el tiempo que lleva hacer todas las operaciones que están fuera del bucle es a y el tiempo que lleva hacer todas las operaciones que están dentro del bucle es b entonces el tiempo total será $a + nb$, siendo que el código de b se ejecuta n veces.

Como dijimos, las operaciones de a son operaciones sencillas, declaraciones de variables, asignaciones, un **`return`**, por lo que este tiempo si pudiéramos medirlo tardaría una determinada cantidad que no depende del tamaño de n. Es decir, es un tiempo fijo, a. Ahora deberíamos ver si las operaciones del ciclo también son independientes del tamaño.

Tenemos la llamada a `fgets()`. La cantidad de operaciones que se ejecuten estará dada por cuántos caracteres ingrese el usuario. Este número es un número acotado, por un lado porque no tiene sentido que ingresara números de más de 10 dígitos, dado que no podemos representar números mayores a 2^{32}, pero también porque el problema está limitado por los 100 caracteres de `buffer[100]`. Si bien cada llamada tardará un tiempo dependiente del número particular que ingrese el usuario para esa iteración, el tiempo está acotado y podemos simplificar que nunca superará un máximo que no depende del tamaño del problema (como mucho depende del tamaño de `buffer`, pero es fijo).

Podemos hacer un análisis similar para la llamada a `atoi()`. Dependerá del largo del número, pero el problema debería suponer no más de 10 dígitos y en el peor de los casos está acotado por los 100 caracteres del arreglo.

Nos queda la llamada a `realloc()`. Ya vimos el algoritmo de `realloc()` en la sección 11.6: Se realiza un `malloc()` del tamaño pedido y si el mismo es positivo se realiza un `memcpy()` del tamaño viejo al nuevo bloque de memoria. Luego se ejecuta un `free()` de la memoria anterior.

En nuestro problema, en cada iteración se ejecuta un `realloc()` de un vector de tamaño i en un vector de tamaño $i + 1$, para poder almacenar el nuevo valor. Esto se va a ejecutar con $i = 0, 1, 2, \ldots, n-2, n-1$. Es decir el tamaño del vector que se redimensiona no es constante en cada una de las iteraciones, si no que a medida que progresemos en la lectura el mismo crecerá cada vez más hasta llegar a n.

La idea que habíamos planteado de que si llamábamos b al tiempo de cada paso del ciclo podíamos estimar el tiempo total como $a + nb$ ya no tiene sentido, porque acabamos de descubrir que b depende de n, o sea es $b(n)$. Si quisiéramos estimar algo deberíamos sumar los tiempos de cada una de las iteraciones con $i = 0 \,..\, n-1$.

Olvidémosnos un rato de a y b y digamos que el tiempo de las operaciones es sencillamente 1. Bueno, el primer ciclo llevará tiempo 1, mientras que el segundo tiempo 2, y así hasta que el último ciclo llevará tiempo n.[3]. El tiempo total de nuestro ciclo entonces será $1 + 2 + \ldots + (n-1) + n = \sum_{i=1}^{n} i$.[4] ¿Podemos resolver esa serie? Vamos a omitir el desarrollo pero se puede probar que:

$$\sum_{i=1}^{n} i = \frac{n^2 + n}{2}.$$

Veremos que es innecesario pero para dejar tranquilo a cualquiera que se haya emocionado con las constantes reintroduzcamos todas las que hagan falta. Digamos que c es el tiempo de las cosas que están dentro del bucle que no son el `realloc()` y que el `realloc()` tarda di, donde d

[3]Después podemos multiplicar esto por una constante arbitraria c y listo, tendríamos el tiempo real... si nos interesara.

[4]Hay un ± 1 según si miramos el tamaño viejo o el nuevo del vector, usemos estos rangos, no va a cambiar el resultado que nos interesa.

es una constante e i el tamaño. Bueno, podemos poner todo junto el tiempo que va a tardar nuestro problema:

$$T = a + nc + d\frac{n^2 + n}{2}.$$

(Estos dos términos que aparecieron no son otra cosa que $nb(n)$ que lo tuvimos que contar uno por uno.)

Si sirviera para algo, podríamos ejecutar el algoritmo con diferentes tamaños y estimar los valores de las constantes para una determinada computadora y predecir el tiempo que tardaría el algoritmo (en esa misma computadora) para un tamaño n cualquiera. En principio no sirve para nada.

Esta expresión a la que llegamos es innecesariamente complicada porque si bien responde la pregunta que nos habíamos hecho incialmente que era *"¿qué pasa si duplico n?"* tiene tantos términos que no nos deja verlo de forma intuitiva. Simplifiquemos, si n es lo suficientemente grande, los términos constantes o lineales no van a incidir mucho en el resultado y finalmente sólo va a pesar el término cuadrático. Es decir, podemos decir que

$$T \approx \frac{d}{2}n^2.$$

Entonces, si para determinado n_1 nuestro algoritmo lleva un tiempo t_1 si tomamos un $n_2 = 2n_1$, es decir duplicamos el tamaño, nuestro algoritmo llevará un tiempo $t_2 = 4t_1$, es decir cuadruplicará el tiempo. ¿Habías adivinado?

El análisis del consumo de memoria es mucho más sencillo, al final se tendrán n unidades de enteros en el heap. Aunque hay que notar que durante la llamada a `realloc()` en un momento coexistirá el vector viejo y el nuevo, por lo que se necesita momentaneamente el doble de memoria para redimensionar.

17.2. Notación $\mathcal{O}$

Lo que presentamos hasta el momento es una introducción muy superficial al tema de la complejidad computacional y no profundizaremos mucho más porque no es un tema de este curso. Lo importante es tomar noción de que los algoritmos se crean para ser ejecutados y que la eficiencia de los mismos se puede cuantificar tanto tomando tiempos en la vida real como realizando análisis teóricos sobre el mismo código. El análisis de los tiempos no nos interesará en principio para vaticinar cuántos segundos, minutos u horas tardará un algoritmo si no para comparar diferentes algoritmos de forma general.

Introduciremos ahora una forma de notación que nos permite esto, comparar algoritmos:

Definición: *Se dice que* $f(x) \in \mathcal{O}(g(x))$ *si* $f(x) \leq mg(x)$ *con* $m > 0$ *y* $\forall x > x_0$.

No pretendemos que se entienda esta definición formal sino para qué sirve. Continuando el ejemplo anterior podemos decir que

$$a + nc + d\frac{n^2 + n}{2} \in \mathcal{O}(n^2),$$

y de hecho fue la simplificación que hicimos oportunamente.

¿Qué nos pide la definición?, nos pide que busquemos una función que acote siempre por encima a mi función, pero sólo a partir de un n mayor a un n_0 arbitrario. Esa fue la simplificación que hicimos cuando dijimos *"si n es lo suficientemente grande"*, a partir de determinado punto mn^2 será siempre más grande que $a + nc + d\frac{n^2+n}{2}$ fijando un m positivo arbitrario.[5]

[5]Tanto m como n_0 arbitrarios pero fijos, si existen entonces puedo aplicar la simplificación.

El objetivo de la notación $\mathcal{O}$ es descartar tanto los términos de menor orden como todas las constantes que tengan que ver con el tiempo concreto. Nuestro ejemplo anterior se simplifica en decir que $T = \mathcal{O}(n^2)$. Ahí debemos leer que T escala de manera cuadrática con respecto al tamaño del problema n. No importa cuánto tiempo tarda, importa de qué manera va a empeorar cuando agrandemos el problema. Si pudiéramos medir un determinado tiempo para un determinado n entonces podríamos estimar cuánto llevaría para 100 veces más elementos.

Esta es la métrica que utilizaremos para clasificar algoritmos, por ejemplo un algoritmo $\mathcal{O}(1)$ es un algoritmo que siempre tarda lo mismo, sin importar el tamaño del problema, un algoritmo $\mathcal{O}(n)$ escala de forma lineal, habrá algoritmos $\mathcal{O}(n \log n)$ que escalarán de forma "cuasilineal", y así. Independientemente de si un algoritmo lineal tarde 10 segundos y otro algoritmo también lineal tarde 50 segundos, para la notación $\mathcal{O}$ ambos serán $\mathcal{O}(n)$ e idénticos, porque lo que nos interesa es el comportamiento al variar el tamaño de la entrada, no si uno es más veloz que el otro.

Más allá de la comparación con otros algoritmos, ¿la complejidad nos puede aportar más información? Supongamos que corremos nuestro algoritmo de lectura de vectores de enteros con 1000 elementos y nos lleva 1 segundo. No importa mucho el número, pero digamos que es un segundo. Lo interesante sería ver que si el vector fuera 10 veces más grande llevaría 100 veces ese tiempo, o sea poco más de un minuto. Y si fuera 100 veces más grande llevaría 10000 veces, lo cual sería casi 3 horas. Y si fuera 1000 veces más grande llevaría 10 días. Y con 10000 veces más llevaría 3 años. Paremos ahí, si leer 1000 enteros nos lleva 1 segundo, leer 10 millones de enteros nos llevaría 3 años. La República Argentina tiene cerca de 50 millones de habitantes, sin importar si 1000 elementos llevan 1 segundo o 1 milisegundo ¿podríamos usar este algoritmo para leer los datos de un censo?[6], ¿es un algoritmo que funciona bien?

El orden de complejidad no sólo nos permitirá comparar algoritmos, en algunos casos incluso ante la ausencia de otro algoritmo nos permitirá descartar un algoritmo para determinadas tareas.

17.3. Búsqueda binaria

Supongamos el problema de dado un arreglo de elementos saber si un elemento particular está o no en el arreglo y cuál es su posición de estar. Esto es una búsqueda.

Podríamos implementar esta función de este modo:

```
1 /* Busca el elemento en un vector de n elementos.
2 Devuelve la posición del elemento o n de no encontrarlo. */
3 size_t buscar(int vector[], size_t n, int elemento) {
4     for(size_t i = 0; i < n; i++)
5         if(vector[i] == elemento)
6             return i;
7     return n;
8 }
```

De forma secuencial comparamos `elemento` con cada uno de los elementos `vector[i]` y si lo encontramos devolvemos su posición.

Si bien la cantidad de operaciones dependerá de `elemento` y su posición en `vector` de forma genérica podemos decir que el algoritmo es $\mathcal{O}(n)$, el tiempo escalará lineal con respecto a la cantidad n de elementos. En principio que el algoritmo sea lineal lo hace mejor que el algoritmo de lectura de enteros cuadráticos que analizamos en la sección anterior, ahora bien, ¿lineal es lo mejor que podemos obtener?

[6] Los censos se realizan cada 10 años, sumar las respuestas del censo de EEUU de 1880 llevó más de 10 años y motivó la adopción de tarjetas perforadas y sumadoras para el de 1890, este hecho marca el inicio de la historia de IBM y del desarrollo de computadoras para resolver problemas.

Siguiendo con los ejemplos cívicos, supongamos que te llaman para ser presidente de mesa en unas elecciones donde en tu mesa hay un padrón de 1000 electores ordenados de forma alfabética. ¿Por cada persona que vaya a votar empezarías a fijarte si es el primero de la lista, si no lo es fijarte si es el segundo de la lista, y así hasta encontrarlo en la posición 861?[7] Hoy en día con la digitalidad se perdieron los índices, pero hasta hace algunas décadas en todas las viviendas había guías telefónicas con los nombres y teléfonos de todos los abonados, o diccionarios y enciclopedias con todas las palabras del idioma ordenadas. Nadie empezaría a buscar por el comienzo para encontrar un Rodríguez.

La mejor forma de encontrar algo es poder descartar rápidamente dónde no lo voy a encontrar. Volviendo al ejemplo del padrón, ¿qué pasa si lo abro exactamente en el medio y me fijo qué elector es el que está ahí? Si justo se da la chance en mil de que sea justo el que está viniendo a votar el problema se terminó, pero lo más probable es que no sea ese el caso. ¿Comparar el nombre de la mitad del padrón con el elector que quiero buscar me da alguna información adicional?

Bueno, si el padrón está ordenado alfabéticamente mirar el elector del medio me indica inmediatamente si me pasé o si todavía no llegué. Esta sencilla pregunta descarta la mitad exacta del padrón. Si ya me pasé, mi elector tiene que estar en la primera mitad, si todavía no llegué tiene que estar en la segunda mitad. ¿Y cómo continúo?, sencillamente puedo hacer lo mismo con la mitad restante, y la mitad de la mitad y la mitad de la mitad hasta que o encuentre a mi elector o me quede sin padrón para partir al medio y no lo haya encontrado.

El algoritmo será algo así como:

```
1  /* Busca el elemento en un vector de n elementos ordenados.
2  Devuelve la posición del elemento o n de no encontrarlo. */
3  size_t busqueda_binaria(int vector[], size_t n, int elemento) {
4      size_t prim = 0;
5      size_t ult = n - 1;
6      while(prim <= ult) {
7          size_t medio = (prim + ult) / 2;
8
9          if(vector[medio] == elemento)
10             return medio;
11         if(vector[medio] > elemento)
12             ult = medio - 1;
13         else
14             prim = medio + 1;
15     }
16
17     return n;
18 }
```

La porción del `vector` que debemos mirar es la que está entre `prim` y `ult`. Ahora bien, sólo miramos el elemento que está en el `medio` entre los dos, y en base a eso reajustamos la porción según qué encontremos.

Es **importante** notar que la documentación ahora dice *"vector de n elementos ordenados"*, eso está imponiendo una precondición muy fuerte: El vector sí o sí tiene que estar **ordenado**. Si no el algoritmo no funcionaría.

¿Y cuántas operaciones realiza nuestro algoritmo? Sabemos que en cada paso se descarta la mitad de los elementos, ¿y cuántas veces itera entonces?

[7] No se trata de tener empatía con la computadora, pero a veces ponerse en el lugar de lo que pretendemos que resuelva nuestro algoritmo y pensar qué haríamos nosotros si realmente tuviéramos que hacer todas esas operaciones es un buen camino para encontrar mejores soluciones.

Para realizar las cuentas de forma sencilla vamos a suponer que $n = 2^k$, es decir que no tenemos una cantidad arbitraria de elementos sino, convenientemente, una potencia de 2. Esto nos va a permitir simplificar las cuentas y vamos a llegar a un resultado que es válido también cuando esto no se cumpla.

Si hacemos esa asunción en el paso 0 de nuestro algoritmo tenemos 2^k elementos. Si descartamos la mitad en cada paso en el paso 1 del algoritmo tendremos $\frac{2^k}{2} = 2^{k-1}$.[8] En el paso 2 tendremos entonces 2^{k-2} elementos y así seguiremos hasta el paso k. En el paso k tendremos $2^{k-k} = 2^0 = 1$ elementos, es decir, habremos partido tantas veces al medio nuestro vector que lo agotamos. Nuestro algoritmo entonces realiza k pasos, pero sabemos que $n = 2^k \implies k = \log_2 n$.

Entonces podemos concluir que nuestro algoritmo es $\mathcal{O}(\log n)$,[9] es decir, es logarítmico con respecto al tamaño de la entrada.

¿Esto es bueno? Supongamos que tenemos un padrón no de 1000 electores si no de los 50 millones de argentinos: $\log_2(50000000) \approx 26$, en apenas 26 iteraciones puedo saber si alguien está o no en el padrón.[10]

17.3.1. `bsearch()`

Similar a la función `qsort()` que presentamos en la sección 8.11.1 la biblioteca provee una implementación genérica de la búsqueda binaria

```
1 void *bsearch(const void *key, const void *base, size_t nmemb,
    ↪ size_t size, int (*compar)(const void *, const void *));
```

donde los parámetros son los mismos que en `qsort()` y se suma `key`, la clave que buscamos. La función devuelve un puntero a la ocurrencia del elemento en el vector o `NULL` de no encontrarlo.

17.4. Lectura del vector

Empezamos la sección hablando del algoritmo de lectura del vector que resultó cuadrático. ¿Podemos mejorarlo? Sí, podemos mejorarlo.

Supongamos el código

```
1  /* Lee enteros de stdin hasta agotar la entrada, devuelve el
2  vector de enteros leídos por el nombre y la cantidad de enteros
3  a través de n. */
4  int *leer_enteros(size_t *n) {
5      int *v = malloc(1 * sizeof(int));
6      if(v == NULL) return NULL;
7
8      size_t memoria_pedida = 1;
9      size_t i = 0;
10
11     char buffer[100];
12     while(fgets(buffer, 100, stdin) != NULL) {
13         if(i == memoria_pedida) {
```

[8] Otra vez ± 1, porque nuestro algoritmo descarta el elemento del medio, pero no cambia el resultado.

[9] Omitimos la base del logaritmo porque un cambio de base es multiplicar por una constante y en notación $\mathcal{O}$ no nos importan las constantes.

[10] Más aún, si estamos asumiendo que nuestra computadora tiene punteros de 64 bits, significa que no puede haber nunca un vector de más de 2^{64} elementos, entonces si pudiéramos tener una computadora con 18 exabytes (2^{64} bytes) de memoria RAM y el mayor vector posible que podríamos cargar ahí la búsqueda binaria aún no iteraría más de 64 veces.

```
14              int *aux = realloc(v, 2 * memoria_pedida * sizeof(int)
                    ↪ );
15              if(aux == NULL) {
16                  free(v);
17                  return NULL;
18              }
19              v = aux;
20              memoria_pedida *= 2;
21          }
22          v[i++] = atoi(buffer);
23      }
24
25      *n = i;
26      return v;
```

empezamos pidiendo un memoria de un tamaño arbitrario pequeño, 1. Ahora ya no pedimos memoria en todas las iteraciones, ahora pedimos memoria sólo cuando agotamos la cantidad que ya pedimos. El quid del algoritmo está en cuánta memoria adicional pedimos cuando nos quedamos sin memoria. Si pidiéramos una suma fija adicional cada vez pediremos esa suma veces memoria pero al analizar la complejidad esto no cambiará el orden. Ahora bien, en nuestra solución estamos haciendo crecer la memoria de forma exponencial[11]. Intuitivamente, si cada vez pedimos el doble de memoria, vamos a converger rápidamente al valor de n.

¿Cuántas veces pedimos memoria en nuestro algoritmo? La primera vez pedimos 1, la segunda 2, tercera 4, 8, 16 y así hasta superar n. Si miráramos esta serie al revés veríamos que, otra vez, la cantidad de pedidos está relacionada con cuántas veces podemos dividir n por dos, es decir, tenemos una cantidad logarítmica de veces que pedimos memoria. Esta es una reducción muy importante con respecto a pedir de forma lineal.

La cantidad de memoria movida por `realloc()` entonces será la sumatoria de cada uno de los tamaños de redimensión $\sum_{i=1}^{k} 2^i = 2(2^k - 1)$, donde k es la cantidad de pasos $\log_2 n$ por lo tanto el resultado es $2(n-1)$. Es decir, termina siendo $\mathcal{O}(n)$. El algoritmo es lineal en cantidad de operaciones.

En cuanto a memoria podría pasar que justo después de redimensionar se alcance el tamaño de la entrada y entonces estemos ocupando el doble de memoria de lo necesario. Si bien estamos descartanto constantes, no deja de ser el doble que el algoritmo anterior.

Podemos emparchar nuestro algoritmo reemplazando la línea del `return v;` por

```
1  int *aux = realloc(v, *n * sizeof(int));
2  if(aux == NULL)
3      // No vamos a descartar todo si ya habíamos podido leer.
4      return v;
5  return aux;
```

redimensionando al tamaño final.

Notar también que así como en el algoritmo cuadrático necesitábamos momentaneamente el doble de la memoria para hacer el `realloc()` acá necesitamos el triple cuando tengamos un vector de n y pidamos $2n$.

Si en vez de tomar 2 como factor de crecimiento tomamos un número menor el derroche de memoria se reduce sin afectar el orden de complejidad que seguirá siendo lineal.

[11] El factor es 2, la duplicamos, pero obtendríamos el mismo resultado si usáramos un número más conservador.

Capítulo 18

Contenedores

18.1. Concepto

En el capítulo 13 presentamos a los tipos de datos abstractos, donde delegamos en un tipo encapsulado la implementación de determinadas funcionalidades. Dentro de los TDAs que podemos implementar para estructurar nuestros programas hay un conjunto que representan una categoría en sí mismo: Los contenedores.

Llamamos *contenedor* a los TDAs que sirven para almacenar objetos y recuperarlos después. Por ejemplo, en el caso más sencillo de tipos (concretos) que ya manejamos un arreglo de enteros permite almacenar objetos de tipo entero en una determinada posición y luego recuperar ese valor entero más tarde.

En el caso de TDAs la idea es que Alan provea toda la infraestructura necesaria para almacenar los objetos que Bárbara necesite guardar. Volviendo al ejemplo anterior, tal vez Bárbara quiere guardar enteros en un arreglo, pero a su vez necesita que ese arreglo sea dinámico, que al crecer lo haga de forma eficiente (ver la sección 17.4), o que los elementos se inserten de forma ordenada y que por lo tanto se puedan encontrar de forma eficiente (ver la sección 17.3), etc. Tal vez ya le estamos pidiendo tanto al vector que queramos en realidad implementar un TDA que gestione todas esas operaciones. Desde el punto de vista de Bárbara sólo nos interesará al TDA pedirle que almacene determinado valor y que nos lo devuelva si lo necesitamos o nos permita consultar si el mismo está o no en la instancia del TDA.

Notar que en el ejemplo anterior Alan implementa un TDA vector dinámico de enteros. Es decir, si bien Alan implementa primitivas de forma opaca para que después Bárbara las consuma, Alan conoce perfectamente los elementos que Bárbara va a almacenar: enteros. Cuando tenemos ese caso hablamos de un contenedor para determinado tipo.

Ahora bien sabemos que la idea de Alan y Bárbara como dos personajes que interactúan entre sí es un cuentito que nos contamos. En la vida real tal vez Alan implementó una biblioteca años antes y a Bárbara le resulta útil y la utiliza sin pedirle ninguna funcionalidad a Alan. Ahora bien, ¿qué pasa si Bárbara en vez de necesitar almacenar un tipo conocido como un entero necesitara almacenar un tipo particular como por ejemplo un TDA cualquiera? Si queremos que realmente Alan pueda implementar contenedores que sirvan a futuro, tendremos que proveer algún mecanismo para que Alan pueda gestionar los objetos que Bárbara necesita guardar sin conocer el tipo de esos objetos. En ese caso hablaremos de un contenedor *genérico*.

18.2. Listas

Empezaremos presentando una interfaz sencilla, la de un tipo que llamaremos “lista”.
Esta lista tendrá las siguientes primitivas:

`crear_vacia()` → `L`: Creará una instancia de la lista y nos la devolverá como un objeto `L`.

`asignar(L, i, x)`: En la posición `i` de `L` guarda el valor `x`.

`obtener(L, i)` → `x`: Nos devuelve `x`, el valor que estaba en la posición `i` de `L`.

`agregar_al_final(L, x)`: Inserta el valor `x` al final de la lista `L`.

`insertar(L, i, x)`: Inserta el valor `x` en la posición `i` de `L`. Ahora bien, todos los elementos que estaban de esa posición en adelante se tienen que desplazar una posición a la derecha, estamos insertando un valor, no asignando como en una de las primitivas anteriores.

`eliminar(L, i)`: Elimina el valor `i` de la lista `L`. Ahora bien, no puede quedar un agujero en la lista, por lo que todos los elementos posteriores a `i` tienen que desplazarse una posición a la izquierda.

Si bien podemos agregarle otras primitivas a nuestra lista, como por ejemplo, que nos diga si la misma está vacía, o cuál es su longitud, o si un elemento se encuentra o no, etc. de momento empezaremos con esta interfaz.

Si bien Alan tiene libertad de acción en cómo implementar esta interfaz sería deseable que todas las operaciones de la misma se pudieran resolver en tiempo constante, es decir, en $\mathcal{O}(1)$. Tal vez esto se pueda, o tal vez no, y tal vez para Bárbara sea más importante que determinada operación sea eficiente y otras no tanto, dependiendo de qué necesita en su aplicación, y esta relación de compromiso entre qué puede ofrecer Alan y qué necesita Bárbara va a motivar que haya muchas maneras difentes de implementar esta interfaz. Incluso puede haber immplementaciones particulares donde alguna de estas operaciones diréctamente no tengan sentido y se omitan.

18.3. Implementación con un arreglo dinámico

Si quisiéramos implementar la lista de enteros una propuesta válida podría ser:

```
1 struct lista_enteros {
2     int *v;      // El vector de n enteros
3     size_t n;    // La cantidad de elementos del vector
4     // Invariante: Si n == 0 <==> v == NULL
5 };
```

No vamos a implementar las primitivas pero deberíamos poder ver que la primitiva `crear_vacia()` debería poder resolverse en tiempo constante $\mathcal{O}(1)$. Lo mismo corre para las primitivas `asignar()` y `obtener()`, en ambos casos pueden resolverse accediendo a un elemento de un vector, lo cual no deja de ser una suma y desreferenciación de punteros (eso sí, primero validemos que el índice provisto sea válido).

¿Qué pasa con `agregar_al_final()`? Si la lista antes de agregar un elemento tiene una determinada cantidad de elementos, luego de agregarlo tendrá un elemento más. En la representación interna que elegimos `v` tiene exactamente `n` elementos, por lo que estamos obligados a hacer un `realloc()` lo cual implica hacer una copia de los elementos previos del vector a memoria nueva. Nuestra implementación va a forzar que esta primitiva se ejecute en $\mathcal{O}(n)$.

¿Esta es la única implmenetación posible? No, por ejemplo, podríamos implementar el tipo de esta forma:

```
1 struct lista_enteros {
2     int *v;          // El vector de n enteros.
3     size_t n;        // La cantidad de elementos ocupados en el
      ↪ vector.
```

```
4       size_t pedidos; // La cantidad de memoria pedida en v.
5       /* Invariantes:
6           n <= pedidos
7           pedidos > 0
8       */
9   }
```

Con una implementación como esta podemos disociar el pedido de memoria de la necesidad de agrandar la cantidad de elementos del vector. Por ejemplo, si iniciáramos pidiendo memoria para 100 elementos en el constructor, las primeras 100 veces que Bárbara inserte un elemento nuevo al final ya tendríamos resuelta la memoria. Recién cuando Bárbara inserte el elemento 101 pagaremos una penalidad por la redimensión de memoria.

Siendo que Alan generalmente precede a Bárbara, lo más probable es que no tenga ninguna idea de si Bárbara va a insertar pocos o muchos elementos en su lista. En el capítulo previo hablamos largo y tendido de las diferentes estrategias para leer un vector de longitud desconocida. La estrategia que Alan implemente cuando `n == pedidos` va a incidir en el orden de complejidad de `agregar_al_final()`.

En este curso no tenemos las herramientas matemáticas para fundamentar lo que vamos a enunciar, pero lo que concluímos en el capítulo anterior de crecer la memoria de forma exponencial puede extrapolarse a este caso. Podemos decir que si Alan hace crecer a `pedidos` de forma exponencial cada vez que se quede sin espacio entonces Bárbara podrá agregar n elementos en tiempo $\mathcal{O}(n)$. Hasta acá no dijimos nada no fundamentado, demostramos esta afirmación en el capítulo anterior. Ahora bien, si Bárbara puede agregar n elementos en tiempo $\mathcal{O}(n)$ entonces podemos decir que agregar un elemento lo hace en $\mathcal{O}(1)$. Esta última afirmación debería hacerte ruido.

Si volvemos a la idea de que Alan arranca pidiendo memoria para 100 elementos está claro que las primeras 100 inserciones de Bárbara se dan en $\mathcal{O}(1)$. Ahora bien, la siguiente inserción tiene que hacer un `realloc()` sobre un vector de tamaño 100, por lo tanto será $\mathcal{O}(n)$. Si duplicamos memoria[1] entonces por 100 inserciones más no haremos nada y a la vez 200 tendremos que hacer una redimensión que costará 200, pero por 200 inserciones no pagaremos costo y así. Bien, en esta sucesión de muchas veces $\mathcal{O}(1)$ y pocas veces $\mathcal{O}(n)$ donde las veces lineales cada vez se espacían más en el tiempo se puede demostrar que cada inserción es de tiempo **amortizado** $\mathcal{O}(1)$.

¿Qué significa lo de amortizado?, que si miramos desde afuera podemos decir que en promedio todo es $\mathcal{O}(1)$, incluso cuando sabemos que cada una determinada cantidad de pedidos Bárbara va a disparar un pedido de redimensión que lleva tiempo lineal. El concepto de amortización viene de la Economía, podríamos decir que cada una de las primeras 100 veces que Bárbara inserta está ahorrando a futuro para pagar todo junto en la centésima inserción. Esa centésima inserción le cobra por lo que no pagó en las 100 anteriores. No vamos a profundizar en el tema, podríamos habernos quedado con la cuenta inicial que hicimos, si n inserciones llevan tiempo lineal, entonces una sola lleva tiempo constante; de hecho intuitivamente debería verse eso, pero es más complejo y en realidad se dice que lleva tiempo amortizado a constante.

Retomando toda esta disgresión, entonces también la primitiva de `agregar_al_final()` puede implementarse en $\mathcal{O}(1)$ (amortizado).

Si abordamos las dos primitivas restantes, `insertar()` y `eliminar()`, veremos que no hay manera de implementarlas mejor que $\mathcal{O}(n)$ con nuestra implementación. El acto de insertar o eliminar en vector implica desplazar una posición todos los elementos restantes del vector. Es decir, tendremos que recorrer los elementos, eso será siempre lineal.

Entonces, resumiendo, con nuestro TDA lista implementado sobre un arreglo podemos obtener complejidades $\mathcal{O}(1)$ para todas las primitivas salvo `insertar()` y `eliminar()` que

[1] Recordemos que podemos tener crecimiento exponencial multiplicando por otro número que no sea 2, pero 2 nos queda cómodo para hacer las cuentas.

serán $\mathcal{O}(n)$.

18.4. Lista genérica

Si tenemos la implementación de la lista de enteros que hicimos en la subsección anterior podemos reemplazar `int` por cualquier tipo de preferencia y tendremos una lista que funciona para cualquier tipo de datos. Ahora bien, si necesitamos la lista para múltiples tipos de datos tendremos que tener muchas versiones que son, básicamente, el mismo código copiado y pegado, y además, necesitamos tener el código fuente disponible para hacer ese cambio.

Como dijimos al comienzo del capítulo, muchas veces queremos que Alan sea capaz de implementar un contenedor que sea agnóstico al tipo de datos que Bárbara va a guardar en él.

Cada lenguaje tiene sus propuestas para poder desacoplar el contenedor del tipo que se guarda en él y en C lo más usual es hacer contenedores que almacenen elementos de tipo `void *`. Como ya sabemos, el tipo `void *` permite guardar cualquier tipo de puntero, y nada más que punteros, por lo que la propuesta para implementar contenedores genéricos implica que Bárbara debe guardar en él punteros, o sea, nuestro contenedor genérico no servirá para guardar enteros y flotantes sino tipos más complejos.

Si bien no se señaló de forma explícita cuando explicamos TDAs (capítulo 13) y modularización (capítulo 14) fue evidente de que ambas cosas funcionan instanciando los objetos del tipo en el heap. Es decir, es natural manejar a los TDAs a través de punteros, incluso llegando al punto de que Bárbara desconoce qué hay almacenado en la memoria que apunta. Entonces, si bien plantear los contenedores genéricos como receptores de punteros a `void` nos limita el almacenamiento de tipos básicos[2], es un tipo razonable porque nos permite almacenar cualquier TDA que construyamos.

Entonces nuestra lista genérica podría implementarse

```
1 struct lista {
2     void **v;
3     size_t n, pedidos;
4 }
```

con el mismo diseño que explicamos en la subsección anterior y donde `v` ahora es un arreglo dinámico de `void *`.

En principio si tenemos implementada la lista con enteros podemos reemplazar las ocurrencias de `int` por `void *` (donde sea necesario) y tenemos implementada nuestra lista genérica. Ahora bien van a surgir detalles que en la implementación de enteros no tenía sentido considerar.

Independientemente de las primitivas que nos interesaban para la lista, sabemos que todo TDA tiene un destructor. Veamos el código del destructor de la lista de enteros:

```
1 void lista_enteros_destruir(lista_enteros_t *l) {
2     free(l->v);      // Libero la memoria de los elementos.
3     free(l);         // Libero la memoria de la estructura.
4 }
```

El TDA, como corresponde libera su memoria asociada. Tanto el vector `l->v` como la estructura `l` es memoria pedida y gestionada por Alan. El vector contiene todos los valores que Bárbara almacenó. Bárbara dijo "almacená un 5 en la posición 3" y Alan almacenó ese 5. Liberar la memoria asociada al vector elimina esa copia del 5 en la memoria gestionada por Alan.

Si vamos al caso genérico:

[2] Si realmente quisiéramos guardar enteros podríamos poner esos enteros en el heap y almacenar `int *`, llegado el caso.

```
1 void lista_destruir(lista_t *l) {
2     free(l->v);
3     free(l);
4 }
```

si `l->v` contiene a los elementos de Bárbara y esos elementos son punteros liberar el vector no libera la memoria asociada a esos elementos, solamente libera el vector en el cual Alan guardaba esas referencias.

Ahora bien, ¿cómo liberamos esos elementos?, podríamos pensar que llamar a `free()` para cada uno de ellos sea una buena idea, pero, ¿realmente los objetos de Bárbara se liberan de este modo? Podemos tomarnos un segundo para mirar nuestro propio destructor, ni siquiera las listas se destruyen con un único `free()`.

Supongamos que sepamos como se liberan los elementos, ¿Bárbara quiere liberarlos al destruir el vector? Por ejemplo, Bárbara tiene una lista con todos los alumnos del curso. De forma temporal se arma una lista con los alumnos que aprobaron el primer parcial. Bárbara ahora tiene dos listas donde los objetos de una son un subconjunto de los objetos de la otra. Si elimina la lista de alumnos que aprobaron el primer parcial, ¿quiere perder esos alumnos de su curso? Veamos que incluso aunque Alan supiera cómo lidiar con la memoria de Bárbara no puede él decidir si destruye los objetos o no.

La conclusión de los últimos dos párrafos es que sólo Bárbara sabe cómo liberar sus objetos y si desea hacerlo. Esa no es responsabilidad de Alan, Alan sólo los almacena, pero no se preocupa de su contenido.

¿Cómo hacemos entonces? Una solución sería exigirle a Bárbara que antes de destruir la lista se preocupe por eliminar sus elementos, si es que hace falta. Esta solución si bien tiene sentido por lo que ya discutimos es incómoda. La otra solución sería que Bárbara le indique a Alan qué hacer con los elementos, sin que por eso Alan tenga que conocerlos. La forma de delegar este tipo de cosas en C es mediante punteros a funciones. Bárbara le va a pasar a Alan el puntero a una función que sepa qué hacer con la memoria de los datos. Esa función pertenece al universo de Bárbara, por lo que va a conocer cómo lidiar con ellos. Alan se va a limitar simplemente a llamarla para cada uno de los elementos almacenados:

```
1 void lista_destruir(lista_t *l, void (*destruir_elemento)(void *))
    ↪ {
2     if(destruir_elemento != NULL)
3         for(size_t i = 0; i < l->n; i++)
4             destruir_elemento(l->v[i]);
5
6     free(l->v);
7     free(l);
8 }
```

La firma del destructor de elementos es `void f(void *)` recibe un elemento de tipo `void *` y lo destruye, los destructores nunca devuelven nada dado que en C no hay forma de recuperarse si hubiera un error de memoria. Notar que si Bárbara no quisiera eliminar los elementos de la memoria (porque ya los tiene en otro lado, porque son estáticos, etc.) puede invocar al destructor pasando `NULL` como función de destrucción, esto le ahorraría a Bárbara tener que construirse una función que no haga nada si quisiera omitir la iberación de los elementos.

Del lado de Bárbara, por ejemplo:

```
1 void liberar_cadena(void *s) {
2     // s es una cadena
3     free(s);
4 }
```

```
5
6  int main() {
7      lista_t *cadenas = lista_crear_vacia();          // Validar!
8
9      char aux[MAX_CADENA];
10     while(fgets(aux, MAX_CADENA, stdin) != NULL) {
11         char *cadena = malloc(strlen(aux) + 1);      // Validar!
12         strcpy(cadena, aux);
13         lista_agregar_al_final(cadenas, cadena);     // Validar!
14     }
15
16     // ...
17
18     lista_destruir(cadenas, liberar_cadena);
19
20     return 0;
21 }
```

En primer lugar si la lista es genérica ¿cómo sabe Bárbara qué elementos tiene? Bueno, esto es sencillo, lo que es genérico es la implementación de Alan. Para Bárbara la lista no es genérica, es una lista de cadenas. ¿Por qué es de cadenas? Sencillamente porque Bárbara guardó cadenas. No importa la implementación de Alan, si Bárbara guarda elementos de diferente tipo adentro de un contenedor lo que va a guardar van a ser punteros, cuando recupere los elementos va a recuperar punteros y es imposible dada una dirección de memoria saber a qué tipo de elemento pertenece esa memoria. Si Bárbara necesita una lista para guardar cadenas guardará cadenas y recuperará cadenas. En el ejemplo incluso la variable donde almacena la lista se llama `cadenas`.

Si la lista es de cadenas, cada elemento de la misma es una cadena y entonces debe liberarse con la función `liberar_cadenas()`. Notar que Bárbara tiene un `char *`, se lo pasa a Alan en la llamada a `lista_agregar_al_final()` que lo recibe como un `void *` y así lo almacena, pero cuando invoca a `liberar_cadena()` esta función sabe que ese `void *` que recibió es en realidad un `char *` y puede gestionarlo de ese modo.

La realidad es que siendo que `liberar_cadena()` se limita a llamar a `free()` y tiene **la misma** firma que `free()`, tranquilamente Bárbara podría haber invocado `lista_destruir(cadenas, free)` y hubiera sido lo mismo.[3]

18.5. Buscar un elemento

Supongamos que Bárbara quiere saber si un elemento se encuentra o no dentro de la lista y recuperarlo.

Si la interfaz fuera:

```
1 void *lista_buscar(const lista_t *l, void *elem) {
2     for(size_t i = 0; i < l->n; i++)
3         if(l->v[i] == elem)
4             return elem;
5     return NULL;
6 }
```

¿estaríamos realmente buscando?

[3] Y si la firma no fuera la misma consultar lo que ya se discutió sobre wrappers en la sección 8.11.1.

Pensemos un segundo el último ejemplo de la lista con cadenas. Imaginemos que el usuario ingresó por `stdin` un listado de nombres y queremos saber si un determinado nombre está o no

```
1 if(lista_buscar(cadenas, "Juan\n") != NULL) // Está
```

Esta función nunca va a encontrar a la cadena, porque la cadena `"Juan\n"` vive en data mientras que las cadenas de la lista viven en el heap y estamos comparando punteros y no contenido. A estas alturas sabemos que para comparar cadenas tenemos que hacerlo por caracteres (o llamando a `strcmp()`).

Entonces, como Alan desconoce el tipo de los datos, es Bárbara quien tiene que proveer la función de comparación.

En la sección 8.11.1 ya vimos que las funciones de comparación en C tienen una interfaz estándar: Reciben los dos elementos y devueven un entero, si ese entero vale 0 es porque los dos elementos son iguales. Si es negativo es porque el primer elemento es menor al segundo y positivo en caso contrario (sí, para sorpresa de nadie exactamente la interfaz de `strcmp()`).

Entonces Bárbara debe proveer la función de búsqueda

```
1 void *lista_buscar(const lista_t *l, void *elem, int (*comparar)(
    ↪ const void *, const void *)) {
2     for(size_t i = 0; i < l->n; i++)
3         if(comparar(l->v[i], elem) == 0)
4             return l->v[i];
5     return NULL;
6 }
```

Más allá de la función de búsqueda notar que cambiamos la devolución. En la primera implementación devolvíamos `elem` en la segunda devolvemos `l->v[i]`. ¿Es indistinto devolver uno o el otro? ¿Si `elem` es el elemento que buscamos, en qué se diferencia de `l->v[i]`?

Acá tenemos que pensar que el elemento que Bárbara está buscando es el de la lista. El elemento que pasa como parámetro a la función es solo algo que sirve para disparar la igualdad en la búsqueda. Ejemplifiquemos, imaginemos que Bárbara tiene una estructura alumno definida como:

```
1 struct alumno {
2     int padron;
3     char nombre[MAX_CADENA];
4     char apellido[MAX_CADENA];
5     enum carrera carrera;
6     // ... y un montón de cosas más
7 };
```

y generó una lista `alumnos` que contiene a todos los alumnos de la facultad.

Luego en nuestro ejemplo Bárbara quiere recuperar la información del alumno con padrón 100000. Entonces puede hacer algo así:

```
1 int comparar_por_padron(const void *a, const void *b) {
2     const struct alumno *aa = a;
3     const struct alumno *ab = b;
4     return aa->padron - ab->padron;
5 }
6
7 // ...
8 struct alumno busqueda = {.padron = 100000};
```

```
9  struct alumno *encontrado = lista_buscar(alumnos, &busqueda,
       ↪ comparar_por_padron);
```

El alumno con el que Bárbara busca es apenas una cáscara vacía que contiene un padrón, el alumno que Bárbara tiene en la lista, y quiere recuperar tiene toda la información completa. Por esto no es indistinto devolver `elem` o `l->v[i]`, el elemento proporcionado no tiene ningún valor más que servir para encontrar al elemento real.

18.6. Interfaz de lista

Omitiendo la búsqueda una posible interfaz para la lista genérica podría ser la siguiente:

```
1  lista_t *lista_crear_vacia();
2  void lista_destruir(lista_t *l, void (*destruir_elemento)(void *))
       ↪ ;
3
4  bool lista_asignar(lista_t *l, size_t i, void *x);
5  void *lista_obtener(const lista_t *l, size_t i);
6  size_t lista_longitud(const lista_t *l);
7
8  bool lista_agregar_al_final(lista_t *l, void *x);
9  bool lista_insertar(lista_t *l, size_t i, void *x);
10 void *lista_eliminar(lista_t *l, size_t i);
```

Notar que todas las primitivas que pueden fallar devuelven `bool`, devolverán `true` en caso de poder realizar la acción y `false` en caso contrario. La primitiva `lista_eliminar()` retira de la lista el elemento en la posición `i` y lo devuelve. ¿Por qué lo devuelve?, porque si tuviera que removerlo de la lista y no lo devolviera tendría que eliminarlo y otra vez necesitaríamos que Bárbara nos dijera cómo hacerlo. Es más sencillo devolverle el elemento a Bárbara y que ella se responsabilice por su memoria.

Se agregó además una primitiva para obtener la longitud n de la lista.

Capítulo 19

Listas enlazadas

19.1. La lista enlazada

La lista enlazada es un contenedor donde cada uno de los datos se almacena dentro de un nodo. A su vez cada uno de los nodos tiene una referencia al nodo *siguiente* de la lista, como si se tratara de los vagones de un tren.

Por ejemplo, para una lista enlazada de enteros podríamos definir el nodo de la siguiente forma:

```
1 struct nodo {
2     int dato;               // El dato que vamos a almacenar
3     struct nodo *sig;       // La referencia al nodo siguiente
4 };
```

Por ejemplo, si tuviéramos los nodos:

```
1 struct nodo a = {1, NULL};
2 struct nodo b = {2, NULL};
3 struct nodo c = {3, NULL};
```

podríamos engancharlos de este modo:

```
1 a.sig = &b;
2 b.sig = &c;
```

y si miramos la lista a partir de `a` veríamos que la misma es una sucesión de 3 nodos con los datos 1, 2, 3 en ese orden.

Volviendo a la declaración de la estructura, en el lenguaje de programación C no puede anidarse una estructura dentro de sí misma, porque haría falta memoria infinita para eso, pero es perfectamente válido incluir dentro de una estructura una referencia a sí misma. Esta referencia, al ser un puntero, tiene un tamaño acotado por el tamaño de los punteros en la plataforma.

Volviendo a la definición de la lista, si cada nodo contiene una referencia a un siguiente nodo y este a su vez contiene una referencia al siguiente, estaríamos ante una sucesión que no termina más. Ampliando entonces en una lista los nodos contienen una referencia al siguiente nodo si es que existe, y una referencia nula en caso de que no haya ningún nodo a continuación.

Notar que así como el nodo `a` define una lista con los elementos 1, 2 y 3, el nodo `b` también define una lista con los elementos 2 y 3, y que esto es válido para cualquier nodo. La lista tiene una estructura recursiva definida en términos de sí misma.

Más allá de que el código anterior sirve de ejemplo para entender cómo se vinculan varios nodos entre sí, si queremos tener una cantidad indefinida de nodos no vamos a tener variables

para cada uno de ellos. Tiene sentido que los nodos vivan en el heap y referenciarlos desde el primer nodo que representa la lista:

```
struct nodo *crear_nodo(int dato, struct nodo *sig) {
    struct nodo *n = malloc(sizeof(struct nodo));
    if(n == NULL) return NULL;

    n->dato = dato;
    n->sig = sig;

    return n;
}

// ...

struct nodo *primero = crear_nodo(1, NULL);
primero->sig = crear_nodo(2, NULL);
primero->sig->sig = crear_nodo(3, NULL);
```

Genera la misma lista que habíamos generado en el ejemplo anterior. Si estás pensando que la expresión `primero->sig->sig` es poco elegante, estás pensando bien. No sólo es una expresión fea, además si quisiéramos insertar 10 nodos en la lista no pararíamos de agregar `->sig` a la misma.

Por como se estructuran las listas, es mucho más sencillo agregar los nodos al principio que al final como hicimos en los dos ejemplos previos. Podríamos hacer:

```
struct nodo *aux = crear_nodo(3, NULL);
aux = crear_nodo(2, aux);
aux = crear_nodo(1, aux);
struct nodo *primero = aux;
```

y tendríamos exactamente la misma lista que antes. Este es un código que podríamos repetir tantas veces como queramos de forma iterativa.

Como dijimos en la introducción, la lista enlazada es un contenedor cuya unidad de almacenamiento es el nodo, donde cada nodo contiene un dato y una referencia al siguiente nodo. Esta definición es la definición matemática abstracta de la estructura de datos.

19.2. Implementación como TDA

Ahora bien, yendo a la implementación, si Alan quisiera encapsular la lista enlazada como un TDA en C se va a encontrar que definir a la lista en función de su primer nodo, como hicimos en los ejemplos hasta el momento, va a implicar que cualquier operación que modifique el primer nodo (sea insertar o eliminar un elemento) haga que la referencia que Bárbara posee tenga que cambiar. Si bien es posible hacer una implementación donde Bárbara tenga una referencia al primer nodo, la misma requiere que Alan reciba punteros al puntero al nodo (o sea, dobles punteros) para poder modificar esta referencia, o que una lista vacía, es decir que no tiene datos, tampoco debería tener nodos por lo que se representaría como un `NULL`.

Para implementar la lista enlazada como un TDA, nos va a quedar mucho más cómodo si Alan le presenta a Bárbara una estructura que funcione como un *wrapper* del primer nodo de la lista:

```
struct lista {
    struct nodo *prim;      // Primer nodo de la lista
```

```
};

typedef struct lista lista_t;
```

Este tipo `lista_t` será la cara visible de la lista enlazada ante Bárbara. Los nodos serán un detalle de implementación que sólo conocerá Alan. No perdamos de vista que lo único que le interesa a Bárbara son sus datos, esa es la idea de contenedor. El resto es problema de Alan.

Y sí, a partir de ahora vamos a llamar "lista" a la lista enlazada, no confundir con la interfaz genérica de lista que presentamos en el capítulo anterior. Si en algún momento necesitáramos referirnos a la lista genérica lo aclararíamos, por omisión cuando hablemos de listas serán listas enlazadas.

Teniendo esa representación interna el constructor de la lista queda:

```
lista_t *lista_crear() {
    lista_t *l = malloc(sizeof(lista_t));
    if(l == NULL) return NULL;

    l->prim = NULL;

    return l;
}
```

Notar que de esta forma la estructura `lista_t` permite que el primer elemento de la lista mute sin por eso modificar la referencia externa de Bárbara. Del mismo modo, Bárbara tiene una referencia no nula incluso cuando la lista está vacía.

Encapsulemos ahora la operación de insertar al comienzo como primitiva:

```
bool lista_insertar_al_principio(lista_t *l, int dato) {
    struct nodo *n = crear_nodo(dato, l->prim);
    if(n == NULL) return false;

    l->prim = n;

    return true;
}
```

La función `crear_nodo()` que ya presentamos será una función auxiliar privada de Alan[1].

19.3. Recorrer la lista

A diferencia de los arreglos, donde podemos acceder de forma sencilla a cualquier elemento con una operación de punteros, en las listas enlazadas no hay manera de acceder a un nodo que no sea recorriendo de forma transversal la lista desde el comienzo.

Por ejemplo, supongamos que Alan quiera imprimir los datos de la lista[2]:

```
struct nodo *actual = l->prim;
while(actual != NULL) {
    printf("%d\n", actual->dato);
    actual = actual->sig;
}
```

[1] i. e. su declaración estará predecedida por `static` y sólo aparecerá en el `.c` (ver la sección 14.5).

[2] Esto no será una primitiva, pero es un ejemplo sencillo para ver la dinámica.

Con variaciones esta será la plantilla que utilizaremos para recorrer los elementos de la lista. En cada paso de la iteración habrá un nodo que será el de interés, utilizaremos para él el nombre `actual`. La elección del nombre de la variable nos va a servir para entender mejor el algoritmo, se recomienda que el nodo con el que trabajaremos se llame así y los demás sean nombrados relativos a él, es decir `anterior`, `siguiente`, etc. Si el nodo `actual` fuera `NULL` será porque alcanzamos el final de la lista y ya no habrá nodos por recorrer. Mientras `actual` exista podremos manipular `l->dato`. El nodo de la siguiente iteración será el siguiente del actual, o sea `actual->sig`.

Utiizando esta plantilla podemos implementar el destructor de la lista:

```
1  void lista_destruir(lista_t *l) {
2      struct nodo *actual = l->prim;
3      while(actual != NULL) {
4          struct nodo *siguiente = actual->sig;
5          free(actual);
6          actual = siguiente;
7      }
8
9      free(l);
10 }
```

Hay ocasiones en las que necesitamos recorrer la lista pero no queremos llegar hasta el final si no, por ejemplo, encontrar el último nodo. Por ejemplo, si quisiéramos insertar un dato nuevo al final, deberíamos insertar un nodo luego del último nodo, por lo que tendríamos que enlazar `ultimo->sig = nuevo`. Este es un caso en el que no nos sirve la plantilla anterior como viene si no que tendremos que modificarla.

Volviendo al algoritmo si para insertar un nodo al final de la lista necesitamos encontrar el último nodo de la misma, entonces esa condición tiene una restricción muy fuerte adicional: Tiene que haber un último nodo, para lo cual tiene que haber nodos. Es decir, no podemos encontrar el último nodo en una lista vacía, dado que una lista vacía no tiene ningún nodo.

La plantilla de recorrido que presentamos previamente funciona en cualquier lista. Cuando empecemos a modificarla tal vez empiecen a aparecer casos particulares que habrá que contemplar. En el ejemplo que estamos dando, insertar un nodo "al final" de una lista vacía es lo mismo que insertarlo al principio.

Entonces:

```
1  bool lista_insertar_al_final(lista_t *l, int dato) {
2      struct nodo *nuevo = crear_nodo(dato, NULL);    // Estará al
           ↪ final => ->sig = NULL
3      if(nuevo == NULL) return false;
4
5      if(l->prim == NULL) {
6          l->prim = nuevo;
7          return true;
8      }
9
10     struct nodo *actual = l->prim;
11     // Cuando este bucle termine, actual será el último nodo de la
           ↪ lista.
12     while(actual->sig != NULL)
13         actual = actual->sig;
14
15     actual->sig = nuevo;
```

```
16
17      return true;
18  }
```

¿Qué hubiera pasado si no hubiéramos abordado el caso particular de la lista vacía previo a empezar la iteración? Notar que la condición de corte del **while** incluye la expresión `actual->sig`, donde en la primera iteración `actual` es `l->prim`. O sea la primera vez que iteremos estaremos evaluando `l->prim->sig`. Esta expresión sólo tiene sentido si `l->prim != NULL` dado que si no estaríamos haciendo algo así como `NULL->sig` lo cual rompería nuestro programa. Entonces si bien el análisis que hicimos previamente nos había indicado que había un caso particular que abordar si la lista es vacía, también debería ser algo evidente analizando el código. Siempre que tengamos que acceder a miembros de punteros a estructuras tiene que estar garantizado que dichos punteros no sean nulos. Si podrían llegar a serlo entonces tendremos un caso particular. En el caso de las listas enlazadas los casos particulares siempre ocurrirán en los extremos mientras que podremos resolver los casos generales con iteraciones genéricas.

19.4. Eliminando nodos

Como acabamos de decir, generalmente cuando pensamos en listas nos interesan particularmente los nodos de los extremos o genéricamente el resto de la lista. Es por eso que implementamos primitivas para insertar al comienzo o al final, y del mismo modo si vamos a pensar en eliminar nodos nos interesará el primero, el último, o alguno genérico del medio.

Empecemos por el principio:

```
1   bool lista_eliminar_primero(lista_t *l) {
2       if(l->prim == NULL)
3           return false;
4
5       struct nodo *primero = l->prim;
6       l->prim = primero->sig;
7
8       free(primero);
9
10      return true;
11  }
```

Otra vez, sólo podremos eliminar nodos si la lista posee al menos un nodo.

Dejamos como tarea la implementación de la primitiva de eliminar el último.

Implementemos ahora una primitiva que elimine un nodo cualquiera. Lo importante de observar es que un nodo pertenece a una lista porque el nodo anterior lo referencia. Es decir para eliminar un nodo de una lista en realidad habrá que modificar el nodo anterior. Y es importante que cuando tenemos hilos de pensamiento como estos le prestemos atención a estos detalles, si tenemos que modificar el nodo anterior entonces el algoritmo que estamos pensando **requiere** que exista un nodo anterior. Otra vez, eliminar el primer nodo de una lista será un caso particular: La primitiva que ya implementamos.

Implementemos una primitiva que elimine la primera ocurrencia de un determinado dato:

```
1   bool lista_eliminar(lista_t *l, int dato) {
2       if(l->prim == NULL) return false;
3
4       if(l->prim->dato == dato) {
5           struct nodo *a_borrar = l->prim;
```

```
6           l->prim = l->prim->sig;
7           free(a_borrar);
8           return true;
9       }
10
11      // Ya sabemos que el primer nodo no es el que tenemos que
            ↪ borrar.
12      struct nodo *anterior = l->prim;
13
14      while(anterior->sig != NULL) {
15          struct nodo *actual = anterior->sig;
16          if(actual->dato == dato) {
17              // Tenemos que borrar a actual
18              anterior->sig = actual->sig;
19              free(actual);
20              return true;
21          }
22          anterior = actual;
23      }
24
25      return false;
26  }
```

Vamos a insistir por tercera vez en algo, si bien cuando analizamos el anterior no dijimos "la lista vacía es un caso particular" si no que sólo identificamos como caso particular si había que eliminar el primero, para identificar si hay que borrar el primero tuvimos que comprobar el valor de `l->prim->dato`. Lo señalamos porque es importante, si necesitamos acceder al dato del primer elemento entonces **requerimos** que haya un primer elemento, entonces que la lista sea vacía es un caso particular. Si bien en este apunte los códigos se ven como algo terminado la programación es un proceso iterativo donde uno va partiendo de ideas generales y atacando los detalles. Sería perfectamente plausible que si fuéramos a implementar este algoritmo primero implementemos la iteración que representa el caso genérico. Luego identifiquemos que esa iteración no puede abordar si el nodo a borrar es el primero de la lista y que luego identifiquemos que necesitamos que la lista no esté vacía. Ese es un hilo de pensamiento más natural que los ejemplos cerrados que estamos presentando acá. Ahora bien, insistimos con esto porque un algoritmo que no tome en cuenta todos y cada uno de los casos particulares está mal y no va a funcionar. Deben ser tenidos en cuenta todos los casos particulares que correspondan para el problema.

Como es habitual hay múltiples maneras de implementar lo mismo. Podríamos demorar la evaluación del caso particular implementando una iteración sobre el nodo actual en vez del anterior:

```
1   bool lista_eliminar(lista_t *l, int dato) {
2       struct nodo *actual = l->prim;
3       sctuct nodo *anterior = NULL;
4
5       while(actual != NULL) {
6           if(actual->dato == dato) {
7               // Tenemos que borrar a actual
8               if(anterior == NULL)
9                   // actual es el primero de la lista
10                  l->prim = actual->sig;
11              else
```

```
12                  // actual es un nodo del medio
13                  anterior->sig = actual->sig;
14
15              free(actual);
16              return true;
17          }
18          actual = actual->sig;
19      }
20
21      return false;
22  }
```

Notar como diferentes algoritmos pueden tener diferentes casos particulares para la misma operación. Lo importante no es que programes la versión más elegante, sencilla o con menos casos particulares. Lo importante es que para la versión que hayas elegido seas capaz de identificar esas condiciones de borde que si no abordás la implementación no funcionará.

Se deja como ejercicio implementar la primitiva que borre todas y cada una de las ocurrencias de un dato[3].

Más adelante veremos cómo se pueden eliminar los casos particulares que aparecen en la manipulación del primer nodo de una lista versus los nodos del medio.

19.5. Listas genéricas

Si bien implementamos previamente el destructor o la eliminación de un dato de la lista enlazada no perdamos de vista que hasta ahora venimos ejemplificando sobre una lista de enteros. Si tuviéramos una lista genérica el tipo de datos será `void *` por lo tanto la declaración del nodo será:

```
1  struct nodo {
2      void *dato;
3      struct nodo *sig;
4  }
```

Todas las primitivas se modificarán en consecuencia, pero nos importa particularmente señalar las primitivas que manipulan datos.

En los ejemplos anteriores hubo primitivas donde a Alan no le importó el contenido del dato, por ejemplo, para insertarlo al comienzo o al final. Por el otro lado hubo primitivas donde Alan sí miró el contenido, por ejemplo para eliminar un nodo dado el dato (`if(actual->dato == dato)`). Y, más importante, hubo primitivas donde Alan no hizo nada de forma explícita con el dato pero si los datos hubieran sido `void *` debería haberles dado un tratamiento particular, esos son los casos que requieren más atención al ser implícitos.

Por ejemplo, ya abordamos para el TDA lista no enlazada genérico que para destruir el TDA en caso de que haya elementos Bárbara tiene que indicar cómo se destruyen. Entonces el destructor de la lista enlazada genérica deberá ser:

```
1  void lista_destruir(lista_t *l, void (*destruir_dato)(void *)) {
2      struct nodo *actual = l->prim;
3      while(actual != NULL) {
4          struct nodo *siguiente = actual;
5
```

[3] Si vamos a hablar de casos particulares notar que si el dato aparece múltiples veces al comienzo de la lista, después de eliminarlo por primera vez tal vez volvemos a estar en el mismo caso particular.

```
6           if(destruir_dato != NULL)
7               destruir_dato(actual->dato);
8           free(actual);
9
10          actual = siguiente;
11      }
12
13      free(l);
14  }
```

En el caso de, por ejemplo, la primitiva de eliminar el primer dato:

```
1   void *lista_eliminar_primero(lista_t *l) {
2       if(l->prim == NULL)
3           return NULL;
4
5       struct nodo *primero = l->prim;
6       void *dato = primero->dato;
7
8       l->prim = primero->sig;
9       free(primero);
10
11      return dato;
12  }
```

Similar a lo que habíamos discutido en la lista no enlazada, si bien podemos pedirle a Bárbara que nos pase una función de destrucción, siendo que estamos ante un único elemento lo más sencillo es devolvérselo y delegarle el problema a ella.

De forma análoga, en la primitiva de eliminar dado un determinado dato tendremos que indicar cómo identificar a ese dato:

```
1   void *lista_eliminar(lista_t *l, const void *dato, int (*
        ↪ comparar_dato)(const void *, const void *));
```

donde la función `comparar_dato` es una función de comparación como ya vimos en la sección 18.5. Entonces la identificación de la ocurrencia del dato en el nodo será algo como `if(! comparar_dato(actual->dato, dato))`.

Insistiendo con cosas que deberían ser evidentes a esta altura del curso: ¿La función anterior es la que elimina la primera aparición del `dato` o borra todas? Deberías poder razonar la respuesta.

Por comodidad retomaremos los ejemplos con la lista de enteros y no la lista genérica.

19.6. Casos particulares

Vamos a presentar una técnica para eliminar los casos particulares que tenemos con la manipulación del primer nodo versus la manipulación de los nodos interiores de la lista. Desde ya se aclara que esta es una técnica avanzada de punteros y que tal vez simplifique los algoritmos pero complejiza el entendimiento del código. Se presenta por un lado por completitud y por el otro porque es una buena aplicación del tema de punteros.

Empecemos identificando por qué ocurre el caso particular. Por ejemplo, en la versión simplificada de la eliminación del nodo teníamos el siguiente código:

```
1   if(anterior == NULL)
```

```
2       l->prim = actual->sig;
3   else
4       anterior->sig = actual->sig;
```

Estamor partiendo en dos condiciones pero lo que hacemos en las líneas 2 y 4 es prácticamente idéntico. ¿Qué es lo que cambia?

La diferencia fundamental entre ambas líneas es que en la primera estamos modificando algo de tipo `lista_t` y en la otra algo de tipo **`struct`** `nodo`. Del mismo modo `lista_t` tiene un miembro `prim` mientras que **`struct`** `nodo` tiene un miembro `sig`. El hecho de que sean estructuras diferentes con miembros que no coincidan hace que ambos códigos sean irreconciliables y que sí o sí haya que abordar el problema como dos casos diferentes desde la implementación.

Ahora bien, si hacemos foco sobre los miembros en cuestión tanto `lista_t->prim` como **`struct`** `nodo->sig` son ambos de tipo **`struct`** `nodo*`. Es decir, ambas estructuras son diferentes, tienen miembros que se llaman distinto, pero en ambos casos lo que modificamos es una referencia a **`struct`** `nodo` (lo cual debería ser evidente, en ambos casos la asignación es `= actual->sig`, o sea, estamos asignando lo mismo, ergo lo que está a izquierda del igual es del mismo tipo).

¿Qué pasa si en vez de operar con las estructuras operamos directamente con los miembros? Es decir apuntamos directamente a `prim` en el caso de las listas y `sig` en el caso de los nodos. Si ambos miembros son de tipo **`struct`** `nodo *` para apuntar a ellos necesitaremos un nivel más de punteros, pero lo importante es que:

```
1  struct nodo **p;
2
3  p = &l->prim;
4  p = &actual->sig;
```

es un código válido y que el puntero `p` puede apuntar indistintamente al primero de una lista o al siguiente de un nodo.

Si lo anterior es válido, entonces es válido también:

```
1  *p = actual->sig;
```

Si `p` estuviera inicializado con el primero de la lista tendríamos `*(&l->prim)= actual->sig`, donde $*$ *cancela* a `&`, y un reemplazo análoga si fuera un puntero a siguiente de nodo, las mismas expresiones de las líneas 2 y 4 que queríamos escribir de forma uniforme.

Entonces, para entrar en calor, implementemos la iteración de Alan imprimiendo nodos en su lista de enteros con dobles punteros:

```
1  struct nodo **p = &l->prim;
2
3  while(*p != NULL) {
4      struct nodo *actual = *p;
5      printf("%d\n", actual->dato);
6      p = &actual->sig;
7  }
```

Si bien podríamos acceder al dato utilizando `(*p)->dato` es más claro si bajamos el nivel de punteros dentro del cuerpo de la iteración.

Teniendo la plantilla en la cabeza ahora podemos implementar la primitiva de eliminación de la primera ocurrencia de un dato:

```
1  bool lista_eliminar(lista_t *l, int dato) {
2      struct nodo **p = &l->prim;
3
```

```
4       while(*p != NULL) {
5           struct nodo *actual = *p;
6           if(actual->dato == dato) {
7               *p = actual->sig;
8               free(actual);
9               return true;
10          }
11          p = &actual->sig;
12      }
13      return false;
14  }
```

Este sería el código de la eliminación eliminando los casos particulares. Si seguimos el código veremos que en la expresión `*p = actual->sig` el puntero `p` tiene uno de dos valores posibles, o `&l->prim` si es la primera iteración o el `&actual->sig` de la última iteración (o sea, el que era actual ya es anterior porque estamos en el siguiente ciclo) lo cual se expande a lo mismo que habíamos escrito cuando desplegamos el código en dos casos particulares en la implementación previa a esta.

Insistimos en lo que dijimos al comienzo. Esta técnica es una aplicación de punteros interesante, que nos permite subsanar el problema que teníamos al tener que modificar estructuras de tipo diferente y la estamos presentando en este apunte como un ejemplo avanzado de punteros. No recomendamos particularmente aplicar esta técnica a los algoritmos de listas, el nivel previo de implementación es más que suficiente aunque implique lidiar con casos particulares.

19.7. Eficiencia

Retomemos las preguntas del capítulo anterior sobre eficiencia en la interfaz genérica de listas, ¿qué podemos concluir al respecto de las listas enlazadas?

La característica principal de la lista enlazada es que al ser, justamente, una estructura enlazada siempre debemos recorrerla para descubrir sus nodos. El único elemento que está disponible de forma inmediata es el primer elemento de la lista.

Por otro lado, al ser los nodos estructuras independientes entre sí agregar nodos de forma intermedia no implica realizar los movimientos de memoria que tendríamos que hacer al operar sobre arreglos. Insertar o eliminar un nodo en cualquier posición sólo implica mover referencias de lugar. Eso sí, primero hay que encontrar qué nodo es el que queremos modificar y, a menos que estemos manipulando el primer nodo, para eso no tenemos otra alternativa que recorrer la lista.

Entonces en prinicipio serán $\mathcal{O}(1)$ las operaciones de creación de la lista vacía y tanto la inserción como eliminación del primer elemento. Ahora bien, la inserción, eliminación e incluso acceso al dato de cualquier otro elemento o incluso conocer la longitud de la lista tendrán complejidad $\mathcal{O}(n)$ porque implicaran recorrer la lista primero.

¿Se pueden mejorar estos órdenes? Sí, guardando información redundante. ¿Qué queremos decir con redundante? Queremos decir que no vamos a aportar información adicional a lo que ya teníamos con la lista definida por la posición del primer elemento, pero que podemos tener cosas precalculadas que nos ahorren tiempo. Por ejemplo, si nos interesara obtener la longitud de la lista en forma $\mathcal{O}(1)$ podríamos tener un miembro en `lista_t` que lleve esa cuenta. Ahora bien será parte de la invariante de representación que ese miembro lleve la cuenta real de nodos, por lo tanto será condición de todas las primitivas mantener ese número actualizado. Almacenar información redundante implica realizar más trabajo en muchos lugares, dependiendo del caso eso podrá ser o no una ganancia.

En una lista enlazada nunca podremos tener referencias a todos los nodos, si hiciéramos eso

perdería el sentido tener una lista enlazada porque necesitaríamos un arreglo para guardar esas referencias, pero puede ayudar tener una referencia al último nodo.

Si incluyéramos las dos cosas que mencionamos podríamos tener una estructura:

```
struct lista {
    struct nodo *prim;
    struct nodo *ult;
    size_t longitud;
};
```

Tener una referencia al último haría $\mathcal{O}(1)$ la operación de insertar al final. Ambos agregados a la estructura pueden mantenerse actualizados sin empeorar la complejidad de las primitivas ya desarrolladas, por lo que son pura ganancia.

Dicho esto, imaginemos que tuviéramos la primitiva que devuelve el i-ésimo elemento de la lista `int lista_obtener(const lista_t *l, size_t i);` como nos proponía la interfaz genérica de listas. Si esa primitiva existiera Bárbara podría estar tentada de operar:

```
for(size_t i = 0; i < lista_longitud(l); i++) {
    int dato = lista_obtener(l, i);
    printf("%d\n", dato);
}
```

Esto no es buena idea. Si lista obtener tiene complejidad $\mathcal{O}(n)$ y estamos haciendo eso para cada uno de los elementos de la lista, entonces iterar una lista tendrá complejidad $\mathcal{O}(n^2)$.

Para iterar una lista necesitamos recorrer nodos, y dado que los nodos son parte de la implementación interna de Alan, sólo Alan puede iterarla. Una solución que puede ofrecer Alan es recorrer la lista y llamar a alguna función de Bárbara con cada dato, por ejemplo:

```
void lista_recorrer(const lista_t *l, void (*visitar)(int)) {
    struct nodo *actual = l->prim;
    while(actual != NULL) {
        visitar(actual->dato);
        actual = actual->sig;
    }
}
```

De esta forma Bárbara puede delegar en Alan realizar alguna operación sobre cada uno de sus datos.

Esta función que presentamos es un poco limitada por lo que vamos a complejizarla un poco. Vamos a reimplementarla no para la lista de enteros si no para la lista genérica y vamos a agregarle más complejidad a la función que nos manda Bárbara y es más fácil mostrar el código que explicar lo que esperamos:

```
bool lista_recorrer(const lista_t *l, bool (*visitar)(void *dato,
    ↪ void *extra), void *extra) {
    struct nodo *actual = l->prim;
    while(actual != NULL) {
        if(! visitar(actual->dato, extra))
            return false;
        actual = actual->sig;
    }
    return true;
}
```

Introdujimos dos cosas nuevas (y una tercera de forma implícita). Por un lado la función `visitar()` devuelve ahora un booleano, Bárbara puede interrumpir la iteración donde quiera simplemente devolviendo `false`. Por el otro agregamos un parámetro adicional en la función. Ese parámetro puede ser cualquier cosa que Bárbara necesite, Alan como lo recibe lo pasa. Si Bárbara no lo necesitara podría ser `NULL` pero puede ser un entero, un TDA, una estructura, cualquier cosa que le dé contexto a su función de visita. Dijimos que había algo implícito que no estaba en la función anterior y es que en este caso, al tratarse el dato de un `void *` Bárbara puede modificarlo y se modificará en la lista, cosa que no pasaba en la lista de enteros. Notar, finalmente, que la función devuelve un booleano que indica si la iteración se completó o no.

Ejemplifiquemos un poco el uso de esta función por parte de Bárbara. Por ejemplo, si Bárbara quisiera volcar su lista en un archivo:

```
bool escribir_en_archivo(void *dato, void *extra) {
    char *s = dato;
    FILE *f = extra;

    if(fprintf(f, "%s\n", s) < 0)
        return false;

    return true;
}

int main(void) {
    lista_t *l = lista_crear();
    lista_insertar_al_principio("estás?");
    lista_insertar_al_principio("cómo");
    lista_insertar_al_principio("amigo");
    lista_insertar_al_principio("Hola");

    // Lista: {"Hola", "amigo", "cómo", "estás?"}

    FILE *f = fopen("archivo.txt", "wt");
    if(! lista_recorrer(l, escribir_en_archivo, f))
        fprintf(stderr, "Hubo un error escribiendo el archivo\n");
    fclose(f);

    lista_destruir(l, NULL);
    return 0;
}
```

¿Cómo sabe la función `escribir_en_archivo()` que su primer parámetro es una cadena y su segundo parámetro un archivo? Lo sabe porque es una responsabilidad de Bárbara. Si Bárbara guardó en su lista genérica cadenas entonces cada dato será una cadena, del mismo modo, si Bárbara pasa como puntero extra un archivo la función se llamará con ese dato. Bárbara tiene que pasar una función consistente con sus datos. Del mismo modo, al destruir la lista Bárbara sabe que sus cadenas viven en la memoria data (ver sección 8.6) por lo tanto son estáticas y no deben ser liberadas, de ahí el `NULL` como parámetro.

Si no tuviéramos el parámetro extra y quisiéramos implementar esta misma funcionalidad tendríamos o que abrir y cerrar el archivo en cada iteración, o utilizar variables globales o similar, y no podríamos abortar la iteración si hubiera una falla.

Otro ejemplo, misma lista pero Bárbara quiere imprimir los primeros 3 elementos de la lista:

```
bool imprimir_n_primeros(void *dato, void *extra) {
```

```
    char *s = dato;
    int *n = extra;

    if(! *n) return false;

    printf("%s\n", s);
    (*n)--;

    return true;
}

int main(void) {
    // ...

    int n = 3;
    lista_recorrer(l, imprimir_n_primeros, &n);

    // ...
}
```

Como se dijo `extra` puede ser cualquier cosa. En ambos ejemplos fue una única variable, pero si Bárbara necesitara pasar múltiples cosas podría crear una estructura ad hoc que contenga todo el contexto que necesite.

19.8. Iteradores

Como ya dijimos en el mundo de los contenedores a Bárbara lo único que le interesa son los elementos que quiere persistir mientras que corre por cuenta de Alan el cómo los almacena. Un contenedor internamente puede almacenar los elementos como considere y la operación de iteración no siempre es evidente. Asímismo dado que son datos de Bárbara a Bárbara puede interesarle iterar sobre ellos y realizar acciones y no siempre una primitiva de recorrido como la que presentamos en la sección anterior es suficiente. Por esto es que existe una categoría de TDAs que son los iteradores. Implementar un iterador para el contenedor será una responsabilidad de Alan, para permitirnos recorrer los elementos del contenedor de forma sencilla.

El iterador, que es un un TDA en sí mismo, forma parte del TDA del contenedor. Es decir, tiene permitido el acceso a la representación interna del contenedor. Y tiene sentido que sea así, justamente estamos proveyendo de un iterador porque con la encapsulación como TDA no podríamos realizar una iteración eficiente.

La versión más sencilla de iterador nos debería permitir recorrer el contenedor con la estructura de un **for** de C, definido en términos de inicialización, incremento y final:

```
iterador_t *iterador;
for(iterador = iterador_crear(contenedor); ! iterador_termino(
    ↪ iterador); iterador_siguiente(iterador)) {
    void *dato = iterador_actual(iterador);
    // ...
}
iterador_destruir(iterador);
```

Podríamos implementar este iterador de forma muy sencilla:

```c
struct lista_iterador {
    struct nodo *actual;
};

typedef strcut lista_iterador lista_iterador_t;

lista_iterador_t *lista_iterador_crear(const lista_t *l) {
    lista_iterador_t *it = malloc(sizeof(lista_iterador_t));
    if(it == NULL) return NULL;

    it->actual = l->prim;

    return it;
}

void lista_iterador_destruir(lista_iterador_t *it) {
    free(it);
}

bool lista_iterador_termino(const lista_iterador_t *it) {
    return it->actual == NULL;
}

bool lista_iterador_siguiente(lista_iterador_t *it) {
    it->actual = it->actual->sig;
    return it->actual != NULL;
}

void *lista_iterador_actual(const lista_iterador_t *it) {
    return it->actual->dato;
}
```

Esta es la funcionalidad básica del iterador. Este iterador sirve, obviamente, para iterar. Este iterador no modifica los elementos de la lista que itera.

También es posible extender la funcionalidad de un iterador para que le permita a Bárbara eliminar el elemento actual o insertar un elemento previo al actual (y poder seguir recorriendo la lista después de realizar esas operaciones).

La interfaz del mismo sería:

```c
// Elimina el dato actual de la lista y lo devuelve. El iterador
   ↪ pasa a apuntar al siguiente en la lista.
void *lista_iterador_eliminar(lista_iterador_t *it);

// Inserta un dato previo al dato acutal. El iterador permanece
   ↪ apuntando al actual.
bool lista_iterador_insertar(lista_iterador_t *it, void *dato);
```

Para implementar estas primitivas hay que tener dominio sobre la lista, porque las operaciones de modificación al comienzo de la misma necesitan acceso a `l->prim`. Además para eliminar el dato actual o insertar un nodo en la posición previa hay que tener acceso al nodo anterior.

Se deja al alumno la implementación de este iterador de lista enlazada con estas operaciones, pero hay dos implementaciones posibles. O guardando en el iterador el `lista_t` original y el

nodo de la iteración pasada, o implementando un iterador que posea únicamente un doble puntero al miembro del actual, como se vio en la sección 19.6.

En el mundo de los contenedores los iteradores son la forma de proveer una recorrida por los elementos sin importar cómo estén estructurados en memoria. En muchos lenguajes de programación la interfaz que tienen los iteradores está totalmente especificada por el lenguaje y todos los contenedores se recorren con las mismas primitivas. Incluso muchos lenguajes de programación implementan una estructura de control llamada *foreach* que permite recorrer cada elemento de un contenedor sin utilizar índices.

Capítulo 20

Otras estructuras enlazadas

20.1. Pilas

Saliéndonos de los contenedores de tipo lista, nos interesa abordar dos tipos de contenedores que se utilizan mucho en la modelización y resolución de problemas. El primero de ellos es la pila.

La pila es un tipo de contenedor donde se almacenan elementos de tal forma de que los mismos se recuperan en el orden inverso en el que se ingresaron. La particularidad de la pila es que sólo podemos ver el último elemento almacenado. Pensar en que tenemos un mazo de cartas donde apilamos[1] una carta sobre otra sobre otra. Si miramos el pilón veremos sólo la última que agregamos y si la retiramos la anteúltima y así.

La interfaz de la pila se define de la siguiente manera:

`crear_vacia()` → `P:` Crea una pila vacía y nos la devuelve como un objeto `P`.

`es_vacia(P)` → `bool:` Devuelve `true` si la pila `P` está vacía.

`apilar(P, x):` Apila el elemento `x` en el tope de la pila `P`.

`desapilar(P)` → `x:` Desapila el elemento `x` del tope de la pila `P` y lo devuelve.

`ver_tope(P)` → `x:` Devuelve el elemento `x` del tope de la pila `P` (pero no lo desapila).

Notar que la interfaz de pilas no provee ninguna forma de conocer la longitud de una pila, sólo si quedan elementos en ella o no. Tampoco hay forma de iterar los elementos, en la pila **sólo** importa el elemento del tope.

Por completitud en inglés la pila se llama *stack*, la operación de apilado se llama *push*, la de desapilado *pop* y la de ver tope *peek*. No es casualidad que llamemos stack al espacio de la memoria donde viven las variables locales de las funciones. Esta zona de la memoria implementa una pila, apilándose variables cuando entramos a una función y desapilándose cuando retornamos de la misma. Obviamente al terminar una función recuperamos el espacio de la última función llamada previa a ella.

Se dice que la pila es una estructura de tipo LIFO, "*last in, first out*", el último dato que entra es el primero que sale.

Al igual que en la interfaz de lista, queremos implementar todas las operaciones de la pila tal que sean $\mathcal{O}(1)$ cada una de ellas.

[1] Es difícil definir una pila sin utilizar palabras que remiten a pilas, por fuera de la ciencia de datos es una estructura cotidiana.

20.1.1. Pila sobre un arreglo

Es posible implementar la pila sobre un arreglo dinámico, de tal manera que la operación de apilar sea la operación de agregar al final.

El tope de la pila estará siempre al final del arreglo.

Las operaciones de desapilado consistirán en devolver el último elemento del arreglo.

Por la dinámica en la que se dan las operaciones LIFO es fácil ver que todas las funciones que pide la interfaz son amigables con arreglos obteniendo la complejidad $\mathcal{O}(1)$ deseada.

20.1.2. Pila sobre una estructura enlazada

Es posible implementar la pila sobre una estructura enlazada tipo lista. En este caso dado que la pila interactúa con el tope tiene sentido que el tope sea el primer elemento de la lista.

Entonces, las operaciones de apilado serán una inserción al principio mientras que las operaciones de desapilado serán operaciones de borrar el primero.

Al igual que con arreglos es evidente ver que todas estas operaciones serán $\mathcal{O}(1)$.

20.2. Colas

El otro contenedor que nos interesa abordar son las colas. En las colas los elementos se almacenan de tal manera que cuando retiro elementos voy a retirar los que hayan entrado primero. Pensemos, por ejemplo, en la fila de un supermercado. Cada cliente va llegando y se pone al final de la fila. Por el otro lado el cajero (o los cajeros) atienden al cliente que está al frente de la fila.

La interfaz de la pila se define de la siguiente manera:

`crear_vacia()` $\rightarrow$ `C:` Crea una cola vacía y nos la devuelve como un objeto `C`.

`es_vacia(C)` $\rightarrow$ `bool:` Devuelve `true` si la cola `C` está vacía.

`encolar(C, x):` Encola el elemento `x` en el tope de la cola `C`.

`desencolar(C)` $\rightarrow$ `x:` Desencola el elemento `x` del frente de la cola `C` y lo devuelve.

`ver_frente(C)` $\rightarrow$ `x:` Devuelve el elemento `x` del tope de la cola `C` (pero no lo desencola).

En inglés a la cola se la denomina *queue*, la operación de apilado se llama *enqueue*, la de desapilado *dequeue* y ver el frente puede ser *front* o *peek*.

La cola se considera una estructura de tipo FIFO, "*first in, first out*", el primero que llega es el primero que se va. Se utiliza principalmente para vincular pedidos que llegan con sistemas que pueden atender consultas limitadas.

20.2.1. Cola sobre un arreglo

La implementación de una cola sobre un arreglo no es una operación inmediata. A diferencia de la pila, donde los elementos se insertaban desde un lado y se retiraban del otro, en la cola los elementos entran de un lado y se retiran del otro. No hay forma de que usar insertar al comienzo y eliminar del final o viceversa garanticen tiempo constante en ambas operaciones.

Cuando se implementan colas sobre arreglos se suele hacer sobre arreglos estáticos y se implementan como *arreglos circulares*. Un arreglo circular es un arreglo en el que si me paso del final vuelvo a empezar por el comienzo. Dicho de otra forma, si un arreglo `v` tiene `N` elementos puedo acceder al elemento i-ésimo como `v[i % N]`. La operación de módulo mantiene el acceso a los índices en el rango $0..N-1$.

La idea de implementar la cola sobre este arreglo es tener un índice para el frente de la cola y la cantidad de elementos encolados. Simplificando a una cola de enteros con un tamaño `N` fijo en tiempo de compilación:

```
1 struct cola {
2     int datos[N];
3     size_t frente;
4     size_t cantidad;
5 };
6
7 typedef struct cola cola_t;
```

Al inicio tanto el frente como la cantidad valen cero.

El frente es la posición del primer elemento de la cola, mientras que la posición del último se calcula como `frente + cantidad`, siempre todo módulo `N`.

La operación de encolado insertará en la posición del último e incrementará la cantidad:

```
1 void cola_encolar(cola_t *c, int dato) {
2     c->datos[(c->frente + c->cantidad++) % N] = dato;
3 }
```

del mismo modo, la operación de desencolado quitará el elemento del frente, lo avanzará y decrecerá la cantidad:

```
1 int cola_desencolar(cola_t *c) {
2     c->cantidad--;
3     return c->datos[c->frente++ % N];
4 }
```

Se puede ver cómo los elementos encolados estarán entre frente y frente + cantidad, moviéndose siempre hacia el final del arreglo. Cuando uno de los índices se caiga del arreglo va a volver a recomenzar.

Está claro que si la cola está implementada sobre un arreglo de `N` elementos entonces está limitada en tamaño. Esto quiere decir que, guardándonos los chistes, se nos puede llenar la cola. Las primitivas previas no están completas sin el chequeo de que la cola se encuentre llena o se haya vaciado.

Es inmediato ver que:

```
1 bool cola_esta_vacia(const cola_t *c) {
2     return c->cantidad == 0;
3 }
4
5 bool cola_esta_llena(const cola_t *c) {
6     return c->cantidad == N;
7 }
```

Debe ser precondición de encolar que la cola no esté llena y debe ser precondición de desencolar que la cola no esté vacía. Las primitivas previas deben ser modificadas para verificar esto.

Las colas se utilizan en muchísimas operaciones de bajo nivel donde se tiene una capacidad de procesamiento limitada y se atienden pedidos a medida que se da abasto para procesarlos. Por ejemplo, el buffer de entrada de una aplicación se puede implementar como una cola. Si la cantidad de caracteres que un usuario ingresó en el teclado es tanta que no se llegan a procesar (por ejemplo, se trabó una tecla) pueden descartarse todos los que superen la cantidad máxima. En muchas aplicaciones de bajo nivel se trabaja con memoria limitada y con buffers estáticos. En

otras aplicaciones si una cola se llena bien se puede redimensionar, copiar todos los elementos viejos a la nueva cola y seguir trabajando. Si las colas se utilizan para ordenar los recursos que llegan en función de los recursos que se pueden atender, muchas veces que se llene una cola implica que hay un problema más grave porque el sistema no da abasto para atender los pedidos que llegan.

20.2.2. Cola sobre una estructura enlazada

A diferencia de la implementación de la cola sobre un arreglo que es complicada, la cola se implementa de manera muy natural sobre una estructura enlazada. Si la estructura enlazada se implementa con una referencia al primer elemento y al último elemento son eficientes tanto las operaciones de insertar al final como de eliminar el primero.

Entonces la cola, a diferencia de la pila, crecerá hacia adelante. El primer elemento de la estructura enlazada será el frente de la cola, mientras que el último de la estructura enlazada será el último de la cola. Desencolar será la operación de eliminar primero, encolar será la operación de insertar al final. Ambas operaciones se realizan en $\mathcal{O}(1)$.

A diferencia de la cola sobre arreglo, la implementada sobre una estructura enlazada no tiene límite de capacidad.

20.3. Otras estructuras enlazadas

Sin interés por ser exhaustivos mencionaremos algunas variaciones de estas estructuras enlazadas y algunas versiones más generales de las mismas.

20.3.1. Listas doblemente enlazadas

La lista doblemente enlazada es una lista tal que cada nodo contiene dos referencias, una al nodo anterior y otra al nodo siguiente.

Esto permite que la lista pueda recorrerse en uno u otro sentido y que sean eficientes tanto las operaciones referidas al primer nodo como al último.

20.3.2. Listas ordenadas

Una lista enlazada ordenada es una lista en la cual los datos están insertados según algún criterio de ordenamiento.

En este tipo no tendremos primitivas de insertar al comienzo, al final ni en ninguna posición particular. La inserción sólo puede realizarse en la posición que no altere el orden.

Más allá de que la lista ordenada tenga aplicaciones, notar que no va a tener las ventajas que podría tener un vector ordenado, dado que no se puede aplicar el algoritmo de búsqueda binaria (ver 17.3) sobre una lista enlazada. El acceso a cualquier elemento va a seguir siendo $\mathcal{O}(n)$.

20.3.3. Listas circulares

En las listas circulares el siguiente del último nodo apunta al primero, es decir, la lista se puede recorrer infinitas veces.

Esta es una estructura que suele utilizarse, por ejemplo, cuando se quiere ciclar de forma indefinida entre diferentes tareas.

20.3.4. Árboles

No vamos a profundizar en árboles dado que es un tema que excede a esta materia, sino apenas describirlos. Los árboles son similares a las listas enlazadas, pero cada nodo tiene en principio referencias a dos nodos (ramas) siguientes, uno a izquierda y uno a derecha. Es decir, es una lista que se bifurca en dos en cada nodo.

Hay diferentes maneras de construir y recorrer los árboles, pero la idea general detrás de los árboles es que si todos los datos se pueden distribuir de manera balanceada entre los nodos a izquierda y derecha de cada nodo, entonces pueden almacenarse N nodos con una distancia desde el primero hasta el final de $\log_2 N$. Esto permite que muchas operaciones que en las listas enlazadas son lineales decrezcan a órdenes de complejidad logarítmicos.

Cuando se dice que en un árbol cada nodo tiene dos ramas estamos hablando de árboles binarios. Se pueden plantear árboles con tantas ramas como uno quiera.

Las listas enlazadas son un caso patológico de un árbol cuyos nodos están tan desbalanceados que tienen sólo un nodo siguiente.

20.3.5. Grafos

Los grafos son estructuras enlazadas donde cada nodo puede contener referencias (aristas) a cualquier otro nodo, sin límites de cantidad. A diferencia de las listas enlazadas y los árboles, donde los nodos no pueden tener referencias a nodos que están antes en la estructura, los grafos pueden tener ciclos, es decir, podemos recorrer las referencias y volver a nodos que ya atravesamos.

Los grafos se utilizan para modelar problemas como por ejemplo un sistema de rutas, o el tendido eléctrico, redes, etc.

Un árbol se puede definir en términos de grafos como un grafo acíclico dirigido. Por lo que las listas enlazadas son, a su vez, un caso particular de grafos.

20.3.6. Colas de prioridad

Cuando presentamos las colas utilizamos de ejemplo la cola de un supermercado. Las colas del supermercado no son colas como las que mencionamos, hay personas que tiene prioridad para adelantarse en la cola, sin por eso perderse el orden de llegada.

Muchos problemas de la vida real se resuelven sobre colas de prioridad, es decir, colas que pueden mantener el orden relativo entre elementos de la misma prioridad pero que tienen capacidad de atender antes a elementos con mayor prioridad. Pensar en la atención de una guardia médica, o en un banco que tiene diferentes categorías de clientes, etc.

Las colas de prioridad generalmente no se implementan sobre listas enlazadas sino sobre árboles.

Capítulo 21

Recursividad

21.1. Recursividad

La recursividad es cuando la definición de un problema se da mediante el uso de recursividad. No, volvamos a empezar. Decimos que la solución de un problema es recursiva cuando se da en términos de sí mismo. Por ejemplo, en los capítulos anteriores estudiamos estructuras enlazadas, ¿cómo es la estructura de una lista?, una lista está formada por nodos donde cada nodo tiene dos cosas: un dato y la referencia a un nodo, que a su vez tiene un dato y la referencia a un nodo, que a su vez tiene un dato y la referencia a un nodo... que finalmente tiene una referencia nula. Si no la lista sería infinita.

Si bien en el mundo abstracto pueden haber definiciones recursivas que sean infinitas, en el mundo computacional tenemos memoria limitada y necesitamos que las definiciones recursivas en algún momento se terminen. Retomando el ejemplo, la lista tiene un *caso general* que es **recursivo**, nodo contiene referencia a siguiente nodo y esta estructura se repite tantas veces como queramos, y finalmente tiene un *caso base* que es **no recursivo** donde se interrumpe la recursividad.

En el mundo de las matemáticas hay muchos problemas que se definen de forma recursiva mediante ecuaciones de recurrencia, por ejemplo la definición del factorial:

$$n! = \begin{cases} n(n-1)! & \text{si } n > 0, \\ 1 & \text{si } n = 0. \end{cases}$$

Por ejemplo, si quisiéramos calcular 4! veremos que, como $4 > 0$ tenemos que operar $4 \times 3!$, lo cual implica calcular un nuevo factorial. Luego $4 \times 3 \times 2!$ que implica calcular $4 \times 3 \times 2 \times 1!$. Y parecería que vamos a seguir operando por siempre, pero resulta que la fórmula anterior implica calcular $4 \times 3 \times 2 \times 1 \times 0!$ y para resolverla tenemos que calcular 0! y este valor no está definido de forma recursiva sino que se trata de un caso base, vale 1. Entonces $4! = 4 \times 3 \times 2 \times 1 \times 1 = 24$.

Si quisiéramos implementar esta función en C no habría mayores complicaciones:

```
1 int factorial(int n) {
2     // Caso base
3     if(n == 0)
4         return 1;
5
6     // Caso general
7     return n * factorial(n - 1);
8 }
```

En la sección 7.2 estudiamos la pila de ejecución de funciones. Ahí vimos que cada invocación de una función genera un marco de ejecución diferente, por lo que el espacio local de variables es distinto en cada instancia.

Si invocáramos a `factorial(4)`, dado que `4` no representa un caso base se ejecutará la línea de **`return`** `n * factorial(n - 1)`. Para resolver esta expresión hay que evaluar la llamada a `factorial(n - 1)`. Llamar a una función implica suspender la ejecución de la función actual y apilar un nuevo espacio de variables en el stack. Mientras la invocación de `factorial(4)` espera se llamará a `factorial(3)` y se repetirá la misma situación. La llamada con 3 invocará a la llamada con 2, que invocará a la llamada con 1 que se quedará esperando a la llamada de `factorial(0)` para operar `n * factorial(n)`. Ahora bien, cuando se invoque `factorial(0)` la misma constituye un caso base, por lo que la función terminará en la expresión **`return`** `1`. Es decir `factorial(0)` evalúa a 1, por lo que la línea **`return`** `n * factorial(0)` de la llamada a `factorial(1)` evaluará a **`return`** `1 * 1` y la función devolverá un 1. Eso desbloqueará la llamada a la función con 2, que devolverá un 2, y a su vez desbloqueará la llamada a la función con 3 que devolverá un 6. Nos habíamos quedando esperando en la llamada original al factorial que operará 4×6 y devolverá 24.

En la última oración del párrafo anterior hablamos de "la llamada original al factorial". La realidad es que, a menos que tengamos acceso a todo el código fuente del programa para analizarlo, nunca podemos asumir que *nuestra* invocación a una función recursiva es la "original". Sí sabemos que para computar el factorial de 4 tendremos que depender de la misma función evaluada en 3, pero nada nos indica que no se está invocando factorial de 4 para computar el factorial de un número superior. Esta es la esencia de un procedimiento recursivo.

21.2. Iteración versus recursión

En la sección anterior dijimos que el factorial se definía de forma recursiva, sin embargo probablemente lo hayas visto definido como $n! = 2 \times 3 \times \cdots \times (n-1) \times n$, o, lo que es lo mismo como $n! = \Pi_{i=2}^{n} i$, donde ambas definiciones son evidentemente iterativas. Entonces, ¿el factorial es recursivo o iterativo?

Los problemas nunca son iterativos o recursivos, esa distinción aplica para los algoritmos. Existe una serie de problemas que tal vez sea más sencillo pensarlos de forma iterativa o tal vez sea más sencillo pensarlos de forma recursiva, pero todo lo que se puede resolver de una forma se puede resolver de la otra, y muchos algoritmos son mixtos.

En este curso vamos a referirnos a un algoritmo o implementación como recursiva cuando el problema se resuelve con una función que se llama a sí misma repetidas veces para ir reduciendo el problema. En cambio diremos que es iterativo cuando lo hace iterando.

Volviendo al factorial, supongamos la siguiente implementación iterativa:

```
1 int factorial(int n) {
2     int f = 1;
3     for(int i = 2; i <= n; i++)
4         f *= i;
5     return f;
6 }
```

Nos interesará comparar cómo se comporta esta implementación con respecto a la recursiva que presentamos en la sección anterior.

Con lo que sabemos deberíamos poder caracterizar fácilmente a la implementación iterativa. Tenemos una iteración que depende del valor de n, dentro de ella se realizan operaciones básicas, por lo que el algoritmo debería ser $\mathcal{O}(n)$ en tiempo de ejecución. Ahora bien, vamos a sumar una variable de análisis que no solemos medir y es el consumo de memoria. Esta función requiere para operar lo que contiene el marco de ejecución de `factorial()`, apenas un par de

variables locales. No importa el tamaño de n, la función siempre va a consumir los mismos recursos espaciales, por lo tanto será $\mathcal{O}(1)$ en uso de memoria.

En la sección anterior hicimos un seguimiento de la ejecución de la versión recursiva. Vimos que la cantidad de llamadas recursivas va a depender del valor de n. A diferencia de la versión iterativa, la función de la implementación recursiva apenas hace operaciones sencillas dentro, no importa el tamaño de n (a menos que justo valga 0 y sea un caso base) la función se va a ejecutar en $\mathcal{O}(1)$ más el tiempo que lleve ejecutar `factorial(n - 1)`. Como la cantidad de veces que se autollama es lineal podemos concluir que la implementación es $\mathcal{O}(n)$ en tiempo.

Ahora bien, ¿qué pasa en espacio? A diferencia de la implementación iterativa, en la versión recursiva cada llamada a la función apila un nuevo marco de ejecución en el stack y las distintas llamadas se irán acumulando hasta llegar al caso base. Entonces en el peor momento vamos a tener aproximadamente n marcos de ejecución en el stack. Cada uno de ellos tiene un tamaño fijo el cual no depende de n, pero sumando a todos ellos el espacio necesario será $\mathcal{O}(n)$.

Comparando la solución iterativa con la recursiva observamos que ambas demoran el mismo orden de complejidad temporal pero la versión recursiva necesita memoria lineal.

Supongamos el algoritmo de cálculo de la potencia x^n en su versión iterativa:

```
1 float potencia(float x, int n) {
2     int p = x;
3     for(int i = 0; i < n; i++)
4         p *= x;
5     return p;
6 }
```

No hay mucho que decir sobre esta implementación, es el algoritmo que aprendemos en la escuela primaria. Aunque sí hay algo que decir, ¿cómo calcularías a mano 3^{11}? Más allá de que la respuesta es $3^{11} = 3 \times 3 \times 3 \times \ldots \times 3$, no sabemos cómo realizar una multiplicación en simultaneo de 11 miembros por lo que si tuviéramos que operar a mano haríamos las cuentas de a pares. Muchas veces como la computadora opera de forma rápida, somos muy veloces para programar cosas que si tuviéramos que hacerlas a mano lo pensaríamos dos veces.

Volvamos al problema:

$$3^{11} = 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3.$$

¿Hay algún resultado que podamos calcular una única vez y reutilizarlo? Probablemente identifiques rápido que calculando una única vez $3 \times 3 = 9$ podemos casi que reducir los términos a la mitad y llegar a una expresión de tipo $(3 \times 3)^5 \times 3$. Como dijimos, cuando tenemos que hacerlo a mano probablemente miremos con atención, nada mal, pasamos de tener que resolver 11 multiplicaciones a tener que resolver sólo 7. ¿Se puede ir más allá? Probablemente podamos hacer con la potencia que apareció lo mismo que hicimos antes.

Habiendo planteado la idea vamos a proponer esta forma de calcular las potencias. Si n es par podemos calcular x^n sencillamente como $x^{\frac{n}{2}} \cdot x^{\frac{n}{2}}$, ¿se ve?, es sencillamente agrupar la mitad de los términos por un lado y la mitad por el otro. Dado que las dos pontencias son iguales la operaremos sólo una vez. ¿Y si n fuera impar?, es lo que nos pasó en el ejemplo anterior, tendremos que compensar.

Entonces:

$$x^n = \begin{cases} x^{\frac{n}{2}} \cdot x^{\frac{n}{2}} & \text{si } n \text{ es par,} \\ x^{\lfloor \frac{n}{2} \rfloor} \cdot x^{\lfloor \frac{n}{2} \rfloor} \cdot x & \text{si } n \text{ es impar,} \\ 1 & \text{si } n = 0. \end{cases}$$

Donde $\lfloor \cdots \rfloor$ es el operador piso, redondear para abajo (truncar). Notar que si vamos a definir nuestra solución como una ecuación de recurrencia necesitamos tener un caso base para que la misma esté completa. En este caso elegimos que si $n = 0$ el problema ya no se define más en términos de sí mismo.

Volviendo al ejemplo, $3^{11} = 3^5 \cdot 3^5 \cdot 3$, donde $3^5 = 3^2 \cdot 3^2 \cdot 3$, donde $3^2 = 3^1 \cdot 3^1$, y donde $3^1 = 3^0 \cdot 3^0 \cdot 3$, siendo $3^0 = 1$. ¿Podríamos haber agregado como condición que si $n = 1$ entonces $x^n = x$? Podríamos, hubiéramos ahorrado alguna cuenta. Por fuera de eso, ¿se ve cómo bajamos notablemente el número de operaciones?

Implementemos esta solución en C:

```
1  float potencia(float x, int n) {
2      if(n == 0)
3          return 1;
4
5      float p = potencia(x, n / 2);
6
7      if(n % 2 == 0)
8          return p * p;
9      else
10         return p * p * x;
11 }
```

Repitamos lo mismo que hicimos con el factorial para las soluciones iterativas y recursivas de la potencia.

La versión iterativa es sencilla, en tiempo tenemos una iteración por lo que es $\mathcal{O}(n)$ mientras que en espacio sólo variables locales por lo que será $\mathcal{O}(1)$.

En la versión recursiva el análisis requiere un poco más de atención, pero estamos ante un resultado ya conocido. Cada llamada recursiva hace operaciones sencillas que no dependen de n y en total tendremos tantas llamadas recursivas como veces que podamos dividir a n por 2, como en el algoritmo de la búsqueda binaria (ver sección 17.3). Es decir, la complejidad temporal terminará siendo $\mathcal{O}(\log n)$. En cuanto a la espacial, otra vez tendrá que ver con cuántas llamadas a función tengamos y es este mismo número, o sea el algoritmo es logarítmico tanto en tiempo como en espacio.

A diferencia de lo que habíamos evaluado en el factorial, la formulación recursiva del problema de la potencia realmente aporta una mejora significativa en tiempo. Es cierto que consume más espacio, pero recordemos que logarítmico es prácticamente lineal[1].

Como bien dijimos, las implementaciones iterativas y recusivas son intercambiables, pero no es tan sencillo implementar el algoritmo de potencia eficiente que acabamos de presentar de forma iterativa.

Volviendo al tema de la memoria, como se mencionó en la introducción de memoria dinámica (capítulo 11), el stack es muy limitado en espacio. Si se realizan demasiadas llamadas recursivas el stack se llenará y se generará un desbordamiento del stack (*stack overflow*). Este error es un error irrecuperable. Un algoritmo recursivo que utiliza espacio lineal es bienvenido sólo si el parámetro del que depende es acotado. Si ese parámetro puede ser arbitrariamente grande es una pésima idea proponer una solución recursiva que utilice memoria lineal, sencillamente se llenará el stack y se romperá la aplicación.

21.3. Diseño de algoritmos recursivos

En muchos casos va a ser el objetivo resolver un problema utilizando recursividad, por lo que queremos aprender una forma de pensar problemas de tal manera de llegar a algoritmos recursivos.

Estructuralmente la idea de un algoritmo recursivo es tener casos bases con soluciones definidas y luego un caso general. Para que el caso general resuelva usando recursividad lo

[1] O, mejor dicho, que dado que n no puede ser nunca mayor que el entero más grande, es decir 2^{32}, entonces $\log_2 n \leq 32$. Nunca haremos más de 32 llamadas.

que se va a hacer es intentar desarmar el problema en problemas análogos más sencillos de los cuales podamos obtener una solución de forma más fácil que para el problema original. A eso nos referiremos más adelante cuando hablemos de "reducir" el problema. Cuando decimos reducir en problemas análogos queremos decir que no estamos cambiando la naturaleza del problema que recibimos, el problema va a ser el mismo pero, por ejemplo, con menos elementos.

Habiendo hecho esa introducción una manera de pensar un algoritmo recursivo es siguiendo estos pasos:

1. Empezaremos planteando problemas donde la solución al problema sea inmediata. Con inmediato queremos decir que no haya que hacer operaciones adicionales para conocer la respuesta. Por ejemplo, si el problema fuera buscar un elemento en un vector, ¿qué pasa si el vector está vacío?, puedo decir de forma inmediata que el elemento no está, sin hacer nada adicional.

2. Lo siguiente que tendremos que plantear es una estrategia de reducción del problema. Por ejemplo, sacar un elemento, partir al medio el problema, descartar alguna parte, etc.

3. Ahora vamos a imaginar que tenemos una función ya implementada que me devuelve la solución para el problema reducido. No importa cómo esté implementada, puede ser iterativa, puede ser recursiva, puede ser mágica. Lo importante es que si pregunto la solución al problema reducido la obtengo.

4. Con la solución del problema reducido, ¿cómo respondo mi problema original? Es decir, en el segundo ítem saqué un elemento, partí al medio el problema, descarté alguna parte... ¿cómo la respuesta que me dio la función del ítem 3 me ayuda a resolver el problema original?

5. Este último ítem va a parecer una formalidad pero es el que le da coherencia al resto. Si reduzco el problema como planteé en el ítem 2, ¿llego eventualmente a uno de los casos inmediatos que planteé en el ítem 1? Si sí, entonces ocurre la magia: La función misteriosa que utilicé en el ítem 3 es la misma que estoy implementando y tengo una solución recursiva.

Sin enredarnos más utilicemos esta receta para resolver un problema.

21.3.1. Ejemplo 1

Supongamos que queremos implementar de forma recusiva la función `bool contiene(const int v[], size_t n, int elem)` que nos dice si un elemento está o no en un vector.

El primer ítem nos dice que planteemos condiciones donde sea inmediata la respuesta. Propongamos una: Si el vector tiene 0 elementos entonces podemos decir con seguridad que el elemento no está en él.

Movámosnos al siguiente ítem, reduzcamos el vector sacándole su primer elemento. Notar que este ítem es absolutamente arbitrario, estamos proponiendo esa forma de reducirlo porque queremos, no es la única y la que elijamos cambiará por completo el algoritmo al que lleguemos.

El tercer ítem nos dice que imaginemos que tenemos una función que si la llamamos `bool b = funcion_magica(v + 1, n - 1, elem);` nos puede decir si el elemento está en el vector después de que le sacamos su primer elemento.[2]

El cuarto punto es cuando realmente resolvemos el problema de forma recursiva: Si podemos aprovechar la respuesta del problema reducido para resolver el problema original todo va a estar bien. Si no, fallaremos en obtener una solución recursiva.

[2] Lamentablemente esta función mágica funciona para vectores de tamaño $n - 1$ pero no de tamaño n...

Entonces, teníamos el vector v de n elementos. Podemos pensar al vector como $v = \{v_0, v_1, v_2, \cdots, v_{n-1}\}$. Nosotros partimos al vector como $v = v_0 \cup \{v_1, v_2, \cdots, v_{n-1}\}$ y sabemos, porque nos lo dijo una función, si el elemento que buscamos está en el tramo final del vector. ¿Esto resuelve nuestro problema? En principio hay dos opciones, si el elemento está en ese tramo, entonces el valor de v_0 es irrelevante, el elemento va a estar en el vector original. ¿Y si no está?, en ese caso es importante verificar si v_0 no es el elemento que buscamos, si fuera entonces estaría en el vector original pero no en el reducido.

Bien, tenemos los 4 primeros ítems de nuestra receta el quinto ítem nos dice que tenemos que chequear si nuestra reducción converje a la solución inmediata que planteamos. Bueno, si a un vector de n elementos le saco elementos de a uno por vez eventualmente llegaré a tener un vector de cero elementos. Entonces esto nos dice que no necesitamos la función mágica porque resolvimos el problema de forma recursiva.

Implementémoslo:

```
1 bool contiene(const int v[], size_t n, int elem) {
2     if(n == 0)
3         return false;
4
5     return v[0] == elem || contiene(v + 1, n - 1, elem);
6 }
```

Notar que escribimos literalmente el desarrollo que hicimos previamente, si vamos al detalle la condición `v[0] == elem` es un caso base, aunque no lo hayamos escrito como uno.

21.3.2. Ejemplo 2

La mejor manera de entender la receta es hacer otro ejemplo... pero con el mismo ejemplo. Es decir, vamos a seguir las reglas de manera diferente para llegar a una solución diferente y ponerlas a prueba.

Empecemos con el mismo planteo, el vector vacío no posee el elemento.

En el ítem 2 habíamos tomado una decisión arbitraria, tomemos otra: Vamos a partir el vector al medio.

Entonces si partimos el vector en dos tenemos dos problemas reducidos podremos llamar:

```
1 bool b_izquierda = funcion_magica(v, n / 2, elem);
2 bool b_derecha = funcion_magica(v + n / 2, n - n / 2, elem);
```

(Sí, esa es la forma de partir un vector en dos "mitades", no nos olvidemos de que no sabemos si n es par o no.)

Tenemos la solución a la pregunta sobre las dos mitades del vector, ¿cómo afecta a la solución del vector completo? La respuesta debería ser inmediata, tanto si `b_izquierda` como `b_derecha` son `true` el elemento está en el vector.

Entonces podemos escribir:

```
1 bool contiene(const int v[], size_t n, int elem) {
2     if(n == 0)
3         return false;
4
5     return contiene(v, n / 2, elem) || contiene(v + n / 2, n - n /
      ↪  2, elem);
6 }
```

¿Sí? Pues no, definitivamente no.

Sin entender mucho de recursividad debería llamarte la atención una cosa, tenemos una implemetación que nos dice si un elemento está o no en un vector y en todo el código fuente no hay en ningún lugar una comparación de `elem` con algo. Es más, el único return explícito que aparece dice `false` y no hay ninguna otra expresión booleana que pueda devolver algo distinto. ¿Qué falló?

Falló que no alcanza con los ítems 1, 2, 3 y 4 de la receta. El ítem 5 no es una formalidad o una conclusión, es algo que tiene que ser verificado.

Si partimos un vector de n elementos sucesivas veces al medio no vamos a llegar siempre a un vector de tamaño 0. Es fácil verlo, si tenemos un vector de un elemento $1/2 = 0$ pero $1 - 1/2 = 1$, se parte en un vector de 0 y otro de 1. Entonces **no** podemos decir que la función mágica sea nuestra función.

Destaquemos esto, si realmente tuviéramos implementada previamente una función que resolviera el problema, podríamos llamarla en la línea 5 de nuestra solución y el problema funcionaría. Lo que no podemos decir es que esa función mágica es la nuestra y convertir nuestra solución en una solución recursiva.

¿Cómo solucionamos el problema? Básicamente el problema que estamos teniendo es que la reducción que planteamos en 2 no converje al caso base que propusimos en 1. Una solución sería cambiar la forma de reducir, por lo que cambiaría por completo el algoritmo, la otra solución sería proponer un nuevo caso base. Podría pasarnos que la forma de reducir que pensamos no vaya para ningún lado útil, pero en este caso es inmediato ver que podemos agregar un caso base que solucione el problema.

Retomemos el ítem 1, si el vector tiene un solo elemento también es inmediato decir si el elemento se encuentra en el vector o no. Basta con chequear que ese único elemento sea o no el que buscamos.

Como dividir n por dos repetidas veces siempre va a converger a 1 entonces cumplimos todos los pasos de la receta:

```
1 bool contiene(const int v[], size_t n, int elem) {
2     if(n == 0)
3         return v[0] == elem;
4
5     return contiene(v, n / 2, elem) || contiene(v + n / 2, n - n /
          ↪  2, elem);
6 }
```

Notar que este algoritmo recursivo es diferente a los que analizamos hasta el momento. En los que habíamos analizado hasta ahora había sólo una llamada recursiva y acá hay dos. Si bien estamos partiendo el problema al medio no es que descartamos una de las dos mitades si no que lo partimos al medio para operar las dos mitades por separado. La cantidad de veces totales que se va a llamar a la función será $\mathcal{O}(n)$. Ahora bien, ¿es esta solución lineal en memoria? No, porque las expresiones en C se evalúan de a términos. Cuando cualquiera de las invocaciones de `contiene()` ejecuten la línea 5 primero se resolverá una de las llamadas y luego la otra[3]. Entonces, como se trata de llamadas disjuntas, hasta no agotarse una de las ramas no se evaluará la otra y como cada una de las ramas depende de la cantidad de veces que n pueda divirdirse por dos la máxima memoria terminará siendo $\mathcal{O}(\log n)$.

Entre la solución de sacar un elemento o partir al medio debería ser obvio que la primera no es viable. Cualquier vector de tamaño considerable nos haría explotar el stack. En cambio esta versión parte al medio, por lo tanto es logarítmica. De todos modos no parecemos haber ganado nada con respecto a la implementación iterativa (que implementamos en 17.3), más allá del ejemplo este no sería un buen problema a resolver de forma recursiva.

[3]Si bien en C las expresiones suelen evaluarse en cualquier orden, en este caso hay un operador `||` y se garantiza que se va a evaluar de izquierda a derecha por el cortocircuito (ver 5.5.4).

21.4. Recursividad de cola

Se define como recursividad de cola (*tail recursion*, TR) a las funciones recursivas cuya última cosa que hacen es evaluar el resultado de la llamada recusiva. De los ejemplos implementados hasta el momento la implementación de factorial lo último que realiza es una multplicación, por lo tanto no es recursividad de cola, similar pasa con la potencia y similar pasa con la búsqueda del elemento en el vector partiéndolo al medio.

La implemetación de la búsqueda del elemento sacando de a un elemento por vez podríamos convertirla en recursión de cola simplemente explicitando el caso base que no consideramos:

```
1 bool contiene(const int v[], size_t n, int elem) {
2     if(n == 0)
3         return false;
4
5     if(v[0] == elem)
6         return true;
7
8     return contiene(v + 1, n - 1, elem);
9 }
```

Notar que no hay una manera inmediata de reestructurar el código de las otras funciones para llegar a que el **return** evalúe únicamente una llamada a función.

¿Por qué son importantes las funciones con recursividad de cola? Porque, aunque no lo parezcan son casi iterativas. Si la función no hace ninguna cosa más después de llamarse a sí misma entonces las variables locales del marco de ejecución local no se vuelven a utilizar.

¿Qué pasa si en vez de generar un nuevo marco de ejecución para la llamada a `contiene(` ↪ `v + 1, n - 1, elem)` de la línea 8 reaprovechamos el marco de la ejecución actual. Sólo tendremos que actualizar `v` para que sea `v + 1` y `n` para que sea `n + 1`. Y si el marco no se duplica y la función se repite al final, básicamente lo que tenemos es una iteración que va a romperse únicamente al alcanzar un caso base.

Podríamos reescribir esto de forma explícita:

```
1  bool contiene(const int v[], size_t n, int elem) {
2      while(1) {
3          if(n == 0)
4              return false;
5
6          if(v[0] == elem)
7              return true;
8
9          v = v + 1;
10         n = n - 1;
11         elem = elem;
12     }
13 }
```

Y se ve que tenemos ahora una versión puramente iterativa que hace lo mismo que la versión recursiva. Hace lo mismo pero ocupando $\mathcal{O}(1)$ en memoria en vez de $\mathcal{O}(n)$, lo cual no es irrelevante dado que cuando implementamos esta función dijimos que "no era viable" porque con vectores de tamaño grande haría explotar al stack.

¿Entonces nos interesa la recursividad de cola porque hace inmediato reescribir de forma iterativa? Para eso programemos iterativo y listo. Por empezar, en el ejemplo de buscar un elemento en un vector **ya** habíamos concluído que la recursividad no aportaba nada a la solución

del problema. Pero por el otro lado lo que todavía no mencionamos es que los compiladores tienen la capacidad de identificar la recursividad de cola.

Cuando prensentamos el tema de CLA dijimos que el GCC tenía más de 5000 parámetros, uno de ellos, `-foptimice-sibling-calls`, activa lo que se llama optimización de la recursividad de cola, llamada TRO por sus siglas en inglés (*tail recursion optimization*). Si bien no es común utilizar este flag de compilación de forma explícita muchos flags se activan de forma implícita al activar otras optimizaciones. Las optimizaciones de velocidad `-O2`, `-O3` y de tamaño `-Os` habilitan la TRO en GCC.

Entonces nos interesa la recursividad de cola porque podemos escribir algoritmos recursivos y dejar que el compilador haga su trabajo e interprete nuestro código como si fuera iterativo[4]. En muchos casos esto puede ser más sencillo de pensar o elegante de implementar.

Por ejemplo retomemos la búsqueda binaria (ver 17.3). Hay que decir que la implementación iterativa de la búsqueda binaria tiene índices arbitarios, uno empieza en 0, otro en `n - 1`, el corte se da cuando `prim <= ult`, no es inmediato ver de dónde salen esos límites. En realidad la búsqueda binaria se piensa de forma recursiva: Recibo un vector ordenado, miro el elemento del medio, en base a eso descarto la mitad del vector y repito el proceso, esto hasta quedarme sin vector.

Podemos implementar

```
1  // Devuelve un puntero al elemento o NULL si no lo encuentra.
2  int *busqueda_binaria(int v[], size_t n, int elem) {
3      // Si nos quedamos sin vector el elemento no está:
4      if(n == 0)
5          return NULL;
6
7      size_t medio = n / 2;
8
9      // Lo encontramos en el medio:
10     if(v[medio] == elem)
11         return v + medio;
12
13     if(v[medio] > elem)
14         // Si está está a la izquieda:
15         return busqueda_binaria(v, m - 1, elem);
16     else
17         // Si está está a la derecha:
18         return busqueda_binaria(v + m + 1, n - m - 1, elem);
19 }
```

Es subjetivo si te resulta más fácil de entender la versión iterativa o la versión recursiva del algoritmo, lo importante es que si en la misma hay recursividad de cola para el compilador son lo mismo.

Si estás atento habrás notado que la implementación de la búsqueda binaria de la sección 17.3 devolvía la posición en el vector mientras que nuestra implementación recursiva devuelve un puntero, hablaremos sobre eso en la próxima sección.

21.5. Wrappers

Supongamos la siguiente función que suma todos los elementos de un vector de forma recursiva:

[4]Valga la aclaración, activar la TRO no necesariamente haga que el compilador identifique y optimice nuestro código, todas las operaciones que se delegan en el compilador son sugerencias.

```
1 int sumar(const int v[], size_t n) {
2     if(n == 0)
3         return 0;
4
5     return v[n - 1] + sumar(v, n - 1);
6 }
```

En esta función estamos reduciendo el problema de a un elemento por vez quitando el último elemento del vector. Para compensar el elemento que quitamos lo sumamos a lo que nos devuelve la llamada recursiva.

Hasta aquí no hay nada nuevo, lo que cabe mencionar es que no tenemos recursividad de cola porque la última expresión que se evalúa antes de hacer el **return** es una suma.

¿Se puede de alguna manera modificar esta implementación para que lo último que se realice es la llamada recursiva? Bueno, una forma de hacer esto es subvertir el orden de las operatorias. La idea sería que en vez de tener un caso base que nos devuelva cero y adicionar el elemento que retiramos luego de llamar a la función, lo que podemos hacer es acumular los elementos que vamos retirando y hacer que el caso base sea el que devuelva esa cuenta.

Esto sería algo así:

```
1 static int _sumar(const int v[], size_t n, int acum) {
2     if(n == 0)
3         return acum;
4
5     return _sumar(v, n - 1, acum + v[n - 1]);
6 }
```

(En breve explicaremos ese **static** que apareció.)

Si bien esta función que implementamos hace lo que describimos que queríamos que hiciera, esta función no es la función original que teníamos. Nosotros queríamos una función que sumara los elementos de un vector, y ahora tenemos una función que tiene un parámetro adicional que no tiene razón de ser en una función que acumula. Es más, ese parámetro tiene que valer exactamente 0 en la primer llamada para que la acumulación sea correcta, pero dentro de la función recursiva no tenemos manera de saber si la instancia que se está ejecutando es la primera de la serie o es una intermedia.

El único que puede invocar a esta función es Alan, y es Alan el que se tiene que encargar de acomodar los parámetros en función del problema que Bárbara espera resolver:

```
1 int sumar(const int v[], size_t n) {
2     return _sumar(v, n, 0);
3 }
```

Ahora bien, ¿es recursiva la función `sumar()`? Vamos a considerar que la función `sumar()` resuelve el problema de forma recursiva, porque utiliza a la función `_sumar()` que ejecuta una recursión. Desde la perspectiva de Bárbara que Alan esté usando o no una función auxiliar no altera el resultado, internamente Alan utiliza recursividad.

La función `sumar()` es un wrapper, concepto que ya vimos en la sección 8.11.1. Un envoltorio que se ocupa solamente de cambiarle la cara a una función que está detrás.

Hay muchos problemas de recursividad en los cuales agregar parámetros que no forman parte del problema hace que la resolución sea mucho más sencilla (independientemente de si logramos tener recursividad de cola o no). En todos los casos la firma de la función que ve Bárbara hacia afuera tiene que ser la firma natural del problema en cuestión y es ilógico exigirle a Bárbara que llame a la función con parámetros adicionales que tienen que valer cosas concretas. Es Alan el que tiene que encargarse de hacer esos ajustes.

Cerrando el ejemplo que quedó incompleto de la sección anterior, habíamos implementado una búsqueda binaria que en vez de devolver un `size_t` devolvía un `int *`, ¿cómo se resuelve? ¡wrapper!

```
size_t busqueda_binaria(int v[], size_t n, int elem) {
    int *p = _busqueda_binaria(v, n, elem); // Esta es la de la
        ↪ sección anterior renombrada.
    if(p == NULL) return n;

    return p - v;
}
```

¿Se hubiera podido implementar la versión recursiva para que devolviera el índice en vez del puntero? Sí, y podés pensarla si querés, bastante seguro que va a ser mucho más complicada que devolver el puntero, que es inmediato, e implementar el wrapper.

Para reforzar cómo agregar parámetros puede simplicarle el trabajo a Alan hagamos otro ejemplo. Decimos que una cadena es *capicúa* si se lee igual de atrás para adelante[5]. Queremos implementar una función recursiva que nos diga si una cadena es capicúa o no. Los casos base son sencillos, si la cadena tiene 1 o menos caracteres tiene que ser capicúa, además si el primer carácter es diferente al último la cadena no puede ser capicúa. En caso contrario sacaremos el primer y último carácter e invocaremos de forma recursiva. Implementemos esto:

```
bool es_capicua(const char *s) {
    size_t n = strlen(s);

    if(n <= 1)
        return true;

    if(s[0] != s[n - 1])
        return false;

    char aux[n - 1];
    strncpy(aux, s + 1, n - 2);
    aux[n - 2] = '\0';

    return es_capicua(aux);
}
```

Lo único que podemos decir positivo de esta implementación es que aplica recursividad de cola... el resto es un espanto. Usa VLAs para cadenas de longitud desconocida (y si no los usáramos tendríamos que usar memoria dinámica), cada llamada tiene que operar un `strlen()` y un `strnpcy()` por lo que el algoritmo terminará siendo $\mathcal{O}(n^2)$ en tiempo. Es más, hay más pasos de iteración en estas funciones que llamadas recursivas en el resto de la implementación. Y, por último, si el compilador no aplicara TRO el algoritmo también sería cuadrático en memoria porque cada marco de ejecución ocupa espacio $\mathcal{O}(n)$. Reiteramos: Un espanto.

Mirar cómo cambia la implentación si la función recibe la longitud de la cadena con la que tiene que tratar:

```
static bool _es_capicua(const char *s, size_t n) {
    if(n <= 1)
        return true;
```

[5]Si se tratara de una frase de múltiples palabras diríamos que es un palíndromo, pero en este caso no vamos a considerar espacios, puntuación, etc.

```
4
5       if(s[0] != s[n - 1])
6           return false;
7
8       return _es_capicua(s + 1, n - 2);
9   }
10
11  bool es_capicua(const char *s) {
12      return _es_capicua(s, strlen(s));
13  }
```

Resuelve el mismo problema con complejidad temporal $\mathcal{O}(n)$ y si el compilador aplicara TRO con $\mathcal{O}(1)$ en memoria.

21.6. Técnicas de diseño de algoritmos

En el mundo de los algoritmos y de la solución de problemas la recursividad es una de tantas técnicas que se emplean para resolverlos.

Los algoritmos recursivos se pueden estudiar mucho más en detalle de lo que se hizo en esta introducción y es tema de cursos más avanzados.

Ahora bien, el diseño de algoritmos de forma recursiva es una de muchas formas de abordar problemas que se aplican en la algoritmia. Hay otras técnicas conocidas de resolución de problemas, en este curso introductorio sólo llegamos a presentar esta, con esto queremos decir que ni recursividad es la única ni tampoco la más importante, hay diferentes técnicas que se aplican a diferente tipo de problemas.

Sirvan estas líneas para poner esto en perspectiva, al igual que con otros temas, como por ejemplo el estudio de la complejidad computacional, este curso llega apenas a presentar lo más superficial sobre el tema.

En el siguiente capítulo veremos un par de algoritmos recursivos que superan ampliamente lo que se puede lograr con algoritmos iterativos.

Capítulo 22

Algoritmos de ordenamiento

22.1. Introducción

En la informática hay muchas ocasiones en la que hay que ordenar datos, incluso a tal punto de que en España llaman "ordenadores" a las computadoras. Más allá de la etimología las primeras máquinas que podríamos asimilar a computadoras, que surgieron a finales del siglo XIX, servían para sumar y para ordenar. Incluso estas máquinas de sumar y ordenar anteceden a las computadoras programables y a las computadoras digitales.

Hay múltiples métodos de ordenamiento y no existe un método universal que sea el mejor para todos los problemas, si no que hay métodos mejores según el tipo de datos que se trate.

En este curso vamos a estudiar cuatro métodos de los que se consideran algoritmos comparativos. Un algoritmo comparativo es aquel que necesita comparar un elemento a con un elemento b para saber cuál de los dos va antes en el conjunto ordenado. ¿Existen entonces métodos que pueden ordenar de forma no comparativa? Sí, si hay información adicional sobre los elementos pueden plantearse métodos no comparativos. Por ejemplo, si calificáramos parciales con notas del 1 al 10 y quisiéramos ordenar todos los parciales de la facultad por nota creciente alcanzaría con hacer 10 pilones, uno para cada nota. Repartiríamos cada uno de los exámenes en el pilón que le corresponda según la nota y cuando terminemos simplemente apilaríamos los pilones según nota creciente. Listo. En una sola pasada ordenamos todos los parciales, obtuvimos un método que ordena, en principio, en $\mathcal{O}(n)$. Y, lo más importante, nunca comparamos los exámenes entre sí.

En el mundo de los algoritmos comparativos nunca vamos a poder obtener un orden de complejidad lineal, es más se puede demostrar que lo mejor que se puede lograr con algoritmos comparativos es $\mathcal{O}(n \log n)$. Ahora bien, los métodos comparativos sirven para ordenar cualquier cosa, no algo particular como los parciales que se catalogan en 10 categorías según su nota.

Otra característica que puede estudiarse en un método de ordenamiento es si funciona *in-place* o no. Los algoritmos in-place pueden ordenar los elementos de un vector sobre el mismo vector, mediante operaciones sucesivas, literalmente in-place significa "en el lugar". Los métodos que no son in-place necesitan memoria auxiliar durante el proceso de ordenado. Esto es relevante, porque si queremos ordenar datos que consumen toda nuestra memoria, o incluso, que no podemos poner en memoria, un método que no sea in-place no nos servirá. Por ejemplo, el método que mencionamos para ordenar parciales, así como lo presentamos, necesita 10 pilas que eventualmente van a tener todos los elementos del vector original para operar por lo tanto no es un método in-place.

Vamos a mencionar una última característica que es la estabilidad. Cuando ordenamos partimos de un vector que tiene los elementos en un determinado orden relativo. Después de ordenar, ¿los elementos que son iguales según el criterio de ordenamiento van a estar en el

mismo orden en el que estaban en el vector original? Volviendo a nuestro ejemplo, supongamos que los parciales estaban ordenados según el orden en el que los alumnos entregaron el examen. Si los apilamos a medida que recorremos los exámenes, los parciales que tengan la misma nota van a apilarse respetando el orden original. Entonces, el algoritmo ese que mencionamos es estable. Que un algoritmo sea estable significa que si ordenamos por un criterio y luego por un criterio diferente no se va a desordenar el primer criterio. Por ejemplo, si ordenáramos los parciales primero por orden alfabético y luego por nota, todos los que se sacaron 1 van a estar juntos ordenados por orden alfabético.

En este curso nos limitaremos a dos métodos de ordenamiento iterativos, selección e inserción, y dos métodos de ordenamiento recursivos, quicksort y mergesort.

22.2. Selección

El método de selección probablemente sea de los más intuitivos de entender. Como en todos los métodos que vamos a presentar partimos de un vector v de n elementos de tipo entero. Esto no significa que podamos ordenar otras cosas, si no que queremos hacer foco en los métodos y no en los datos. El método lo que propone es empezar haciendo una búsqueda del menor elemento del arreglo. ¿Qué posición debería ocupar ese elemento en el arreglo ordenado? Obviamente la primera posición. Entonces lo que haremos será intercambiar de lugar el menor elemento con el elemento que esté en v_0. El vector entre v_1 y v_{n-1} todavía se encuentra desordenado, ¿cómo seguimos?, volvemos a hacer una búsqueda de mínimo sobre ese resto de vector y esta vez intercambiaremos el mínimo elemento con el de la posición v_1. Y seguiremos haciendo esto hasta que quede un único elemento al final del vector y el vector ya estará ordenado.

La implementación del ordenamiento por selección es muy sencilla, en primer lugar tenemos las dos operaciones que mencionamos: Búsqueda de mínimo e intercambiar dos elementos de lugar:

```
1  // Devuelve un puntero al mínimo elemento del vector v.
2  int *minimo(int v[], size_t n) {
3      int *min = &v[0];
4      for(size_t i = 1; i < n; i++)
5          if(v[i] < *min)
6              min = v + i;
7      return min;
8  }
9
10 // Intercambia el contenido de los punteros a y b.
11 void swap(int *a, int *b) {
12     int aux = *a;
13     *a = *b;
14     *b = aux;
15 }
```

Luego, el método de ordenamiento busca el mínimo y lo mueve a la primera posición reduciendo el vector en uno cada vez:

```
1  // Ordena el vector v mediante el algoritmo de selección
2  void seleccion(int v[], size_t n) {
3      for(size_t i = 0; i < n - 1; i++) {
4          int *min = minimo(v + i, n - i);
5          swap(v + i, min);
```

```
6       }
7   }
```

Notar que al inicio de cada paso de la iteración i, el vector en el rango $0..i$ se encuentra ordenado y más aún todos los elementos $v[j]$ tal que $j < i$ ya están en su posición definitiva. Esa es la invarante de ciclo (ver 12.5) de este método de ordenamiento.

En cuanto a la estabilidad, podemos ver que como nuestra función de mínimo para dos elementos iguales reconoce como menor el primero que haya encontrado, entonces se va a preservar el orden original entre los elementos de v. Por lo tanto se dice que el método es estable[1].

22.2.1. Eficiencia

Antes de entrar en análisis de eficiencia podemos hacer una observación sobre el método: ¿Qué hace este método de ordenamiento si el vector que recibe **ya** está ordenado? Notar que al algoritmo no le importa cómo están los elementos. Si el vector estuviera ordenado el primer elemento va a ser el menor, pero igualmente hacemos una búsqueda de mínimo en todo el vector. Notar que el método este es ciego a cualquier particularidad del vector, siempre va a hacer lo mismo. Cuando analicemos la eficiencia veremos que la complejidad va a ser la misma sin importar ninguna característica en el orden previo de los datos.

La eficiencia temporal del método debería ser fácil de ver. Hacemos $n - 1$ búsquedas de mínimo en un vector. La primera vez sobre los n elementos del mismo, la siguiente sobre los $n - 1$ restantes y así. Ya nos hemos topado con esta serie cuando presentamos el tema de complejidad y sabemos que $\sum_{i=1}^{n} i = \frac{n^2+n}{2}$. Entonces temporalmente nuestro método es $\mathcal{O}(n^2)$.

Podemos hilar un poco más fino en este resultado y hacer una observación, si bien la cantidad de operaciones de comparación y de iteración es cuadrática, la cantidad de movimientos de memoria es lineal, porque sólo se hace un movimiento de elementos por iteración. El método es ineficiente, pero no es muy intensivo en memoria.

En cuanto al comportamiento espacial, el método es in-place, no necesita memoria adicional para ordenar.

22.3. Inserción

Antes de ir al método de ordenamiento supongamos el siguiente problema. Tenemos un vector ordenado y queremos agregar un elemento en él. Ahora bien queremos que el vector siga estando ordenado después de agregar ese elemento, por lo que no podemos agregarlo en cualquier lugar. Algo que podemos hacer es recorrer el vector para encontrar en qué posición debería ir el nuevo elemento y luego *hacerle lugar* para insertarlo, es decir, desplazar todos los elementos que vayan a continuación una posición a la derecha. Otra cosa que podemos hacer es poner el elemento nuevo temporariamente al final del vector. Si el elemento es mayor que el último elemento ya estamos, pero si no, podemos invertir el último (que es el elemento que agregamos) con el anteúltimo. Ahora bien, puede que el elemento anterior al cual pusimos nuestro elemento siga siendo mayor en ese caso volveremos a invertir las posiciones y así hasta que o lleguemos al comienzo o encontremos un elemento menor.

Implementemos esto:

```
1  // Inserta un elemento en un vector ordenado v de n elementos.
2  // Asume que hay lugar para un elemento más en la memoria.
3  void insertar_ordenado(int v[], size_t n, int elem) {
```

[1] Al menos en esta implementación si hubiéramos puesto `if(v[i] <= *min)` en la línea 5 de la función `minimo()` ya no lo sería.

```
4       size_t i;
5       for(i = n; i > 0 && elem > v[i - 1]; i--)
6           v[i] = v[i - 1];
7       v[i] = elem;
8   }
```

Notar que si bien dijimos que íbamos a poner el elemento nuevo al final, lo pusimos "virtualmente", en realidad lo que hicimos fue ir desplazando elementos hacia el final hasta que llegamos a la posición que necesitábamos, habiéndole hecho lugar, y ahí insertamos. Una mezcla entre los dos algoritmos que mencionamos. Pero lo importante de esta implementación es que encontramos la posición a medida que vamos desplazando, es decir, no tenemos que hacer una búsqueda previa de dónde terminará nuestro elemento.

El método de ordenamiento por inserción parte de esta misma idea. Tenemos un vector de n elementos y vamos a ir incorporando al vector los elementos de a uno por vez, haciendo una inserción ordenada de cada uno de ellos:

```
1   // Ordena el vector v según el algoritmo de inserción
2   void insercion(int v[], size_t n) {
3       for(size_t = 1; i < n; i++)
4           insertar_ordenado(v, i, v[i]);
5   }
```

(Nos salteamos el primer paso dado que un vector de un único elemento ya está ordenado.)

Notar que en cualquier paso i todos los elementos previos del vector estarán ya ordenados, esta es su invariante de ciclo, por lo que podemos llamar a la función `insertar_ordenado()`. El método lo que va haciendo en cada iteración es desplazar el elemento actual tantas veces hacia el inicio como sean necesarias para que el vector vuelva a estar ordenado.

Otra vez, como cuando encontramos un elemento *no mayor* interrumpimos el desplazamiento hacia atrás, el método preservará el orden previo para elementos iguales y será entonces estable.

22.3.1. Eficiencia

Empecemos, igual que antes, preguntándonos qué hace este método con un arreglo ordenado. Si el arreglo está ordenado, cada vez que incorporemos un elemento al vector ya será más grande que el último, por lo tanto la inserción ordenada ubicará el elemento al final. Entonces en una única pasada habremos "ordenado" el vector. El algoritmo de inserción tiene complejidad temporal $\mathcal{O}(n)$ si el vector ya está ordenado. Es decir, no pierde tiempo ordenándolo de nuevo.

Dicho esto, ¿y cómo será el caso general, cuando no podamos decir nada del orden previo de los elementos del vector? A menos que estemos en el caso anterior, la función `insertar_ordenado()` tiene complejidad computacional $\mathcal{O}(n)$, no importa si el elemento queda en la mitad, casi al principio o al final, la cantidad de operaciones será proporcional con el tamaño del vector en el que insertemos.

Entonces la complejidad del ordenamiento por inserción será la suma de todas las llamadas a `insertar_ordenado()` con los diferentes tamaños desde 1 hasta n. Es la misma serie que presentamos para selección y ya sabemos que es $\mathcal{O}(n^2)$.

Espacialmente estamos ante otro método in-line, por lo que la complejidad espacial será $\mathcal{O}(1)$.

Notar que, a diferencia del algoritmo de selección, ahora estamos haciendo hasta i movimientos de memoria por iteración por lo que la cantidad de escrituras será cuadrática también. Es más, si el vector original está ordenado al revés haremos exactamente $\frac{n^2+n}{2}$ escrituras de memoria.

22.3.2. ¿Qué pasa en el medio?

En la sección anterior concluímos que para vectores ordenados el algoritmo era lineal y para desordenados era cuadrático. Si recordamos de la primera sección de este capítulo dijimos que lo mejor que se podía obtener con un método comparativo era $\setminus \log n$ y para el método de inserción encontramos dos cotas, una mucho mejor que la ideal y otra mucho peor. ¿Cuándo convendrá usar este método?

Si el vector está ordenado concluímos que en una única pasada el método termina. Ahora bien, ¿qué pasa si hay un único elemento desordenado? En ese caso haremos, además de esa pasada que es inherente a recorrer el vector, una pasada de `insertar_ordenado()` para acomodar ese elemento desordenado en su lugar. O sea, iteraremos dos veces lineal. ¿Si hubiera un elemento más desordenado? tres veces lineal, y así hasta que estén todos desordenados y tendremos n veces lineal, que es el $\mathcal{O}(n^2)$ que habíamos concluído antes.

Mientras el vector tenga pocos, muy pocos (más especificamente, menos de $\log_2 n$) elementos desordenados, el método será todo lo bueno que puede ser un método comparativo. Si tuviera más tenderá a ser un método cuadrático.

Antes de cerrar en esta conclusión, todavía hay más. En el desarrollo anterior asumimos que pocos elementos desordenados se tienen que desplazar mucho por el vector. ¿Pero qué pasa si los elementos en el vector están más o menos ordenados?, es decir, ¿qué pasa si para ubicar a un elemento en su posición definitiva no tengo que moverlo más que un par de posiciones de su posición original? En ese caso la complejidad de `insertar_ordenado()` no será lineal si no que será una constante, y el orden total volverá a dar lineal.

Hay muchos problemas donde tenemos que ordenar cosas que están más o menos ordenadas. Es lo que suele pasar cuando actualizamos datos que están ordenados. Por ejemplo, si tuviéramos un ranking de cualquier cosa: fortunas, posiciones en un torneo, universidades, etc. y actualizáramos los datos de los individuos sería raro que alguien que en el ranking viejo estuviera primero en la tabla luego de la actualización pasara a estar último. Lo más probable es que entre actualización y actualización se quede en el lugar o se mueva un par de posiciones en uno u otro sentido.

El algoritmo de inserción funciona muy mal para ordenar vectores genéricos, pero tiene un papel aceptable si ordenamos vectores que tienen pocos elementos desordenados o si ordenamos vectores que están poco desordenados.

22.4. Mergesort

El mergesort es un método de ordenamiento recursivo. La propuesta del método es partir el vector a ordenar en dos mitades y ordenar ambas mitades de forma recursiva. Para que el problema esté completo, el método tiene que juntar los dos vectores ordenados en uno solo ordenado. La fortaleza del método está puesta en cómo se realiza esta operación de volver a juntar las mitades ordenadas.

22.4.1. Merge

Se llama *merge* al algoritmo que une dos vectores ordenados para obtener un nuevo vector ordenado. ¿Podemos hacer esta operación de forma eficiente, es decir sin volver a ordenar los vectores?

La idea es la siguiente, si tenemos dos subvectores y queremos juntarlos en un vector, todos ellos ordenados, el primer elemento del vector tendrá que ser el menor de los elementos de los subvectores. Ahora bien, si los dos subvectores están ordenados entonces el menor elemento de cada uno de ellos tendrán que estar al comienzo, por lo que el primer elemento del vector tiene que ser el primer elemento de uno de los dos subvectores. Si "sacamos" ese elemento

del subvector que corresponda y lo "pasamos" a la primera posición del vector ahora tenemos que buscar qué elemento es el que va segundo en el mismo. La situación se vuelve a repetir, habiendo sacado el elemento más chico ahora el segundo elemento más chico tiene que ser uno de los que están al comienzo de alguno de los subvectores. Iterativamente iremos sacando el elemento menor entre el primero de los dos subvectores y pasándolos al vector hasta que no queden más elementos en alguno de los dos subvectores. Cuando lleguemos a esa situación todos los elementos que quedan en el subvector que no se haya terminado son mayores que todos los que están en el vector y además están ordenados, por lo que su ubicación final será al final del vector.

Implementemos este algoritmo:

```
1  // Une los dos vectores ordenados a y b y devuelve el vector
   ↪ resultante.
2  // El vector resultante mide na + nb.
3  int *merge(const int a[], size_t na, const int b[], size_t nb) {
4      int *r = malloc((na + nb) * sizeof(int));
5      if(r == NULL) return NULL;
6
7      size_t ia = 0, ib = 0, ir = 0;
8
9      // Iteramos hasta que se termine uno de los dos vectores:
10     while(ia < na && ib < nb) {
11         if(a[ia] < b[ib])
12             // El elemento más chico está al comienzo del
                   ↪ subvector a
13             r[ir++] = a[ia++];
14         else
15             // El elemento más chico está al comienzo del
                   ↪ subvector b
16             r[ir++] = b[ib++];
17     }
18
19     // Si llegamos acá es o porque ia == na o porque ib == nb.
20     // Sólo se ejecutará uno de los siguientes whiles:
21     while(ia < na)
22         r[ir++] = a[ia++];
23
24     while(ib < nb)
25         r[ir++] = b[ib++];
26
27     return r;
28 }
```

En la implementación los índices i_a, i_b e i_r sirven para saber el primer elemento de cada uno de los subvectores y del vector resultado respectivamente. En cada paso de la iteración incrementamos el índice de un único subvector. Es importante notar que, si por ejemplo, en la primera iteración el elemento más chico era el primero del subvector a eso no aporta nada de información a cuál va a ser el elemento menor de la siguiente iteración, puede ser el de b o puede ser de nuevo el de a. Incluso podría llegar a pasar que todos los elementos de a sean menores a los de b, en ese caso se entrará siempre al `if` y recién cuando se termine la iteración principal se copiarán todos los elementos de b al final de r.

Lo importante a ver del algoritmo de merge es que podemos realizar la fusión de dos subvectores ordenados en un nuevo arreglo en una sola pasada sobre ambos arreglos, es decir

la complejidad temporal del método será $\mathcal{O}(n_a + n_b)$ o, lo que es lo mismo, $\mathcal{O}(n_r)$.

En cuanto a la complejidad espacial, no hay manera de realizar la operación de merge si no es utilizando un tercer vector, por lo que también la complejidad espacial será $\mathcal{O}(n_a + n_b)$.

22.4.2. Mergesort

Habiendo ya resuelto la operación de merge, que nos permite juntar dos arreglos ordenados en un arreglo entonces podemos implementar el algoritmo del mergesort.

Como estamos partiendo el problema al medio consideraremos como casos bases que el vector tenga uno o menos elementos, un vector de 0 o 1 elemento ya está ordenado.

Entonces:

```
int *_mergesort(const int v[], size_t n) {
    if(n <= 1) {
        int *r = malloc(sizeof(int));
        memcpy(r, v, n * sizeof(int));
        return r;
    }

    int *a = _mergesort(v, n / 2);
    int *b = _mergesort(v + n / 2, n - n / 2);

    int *r = merge(a, n / 2, b, n - n / 2);

    free(a);
    free(b);
    return r;
}
```

Notemos que esta implementación es incompleta, se omiten las validaciones de memoria para no complejizarla. Notar además que en el caso base estamos siempre pidiendo memoria para un entero, esto es porque si hiciéramos `malloc(n * sizeof(int))` devolvería `NULL` cuando $n = 0$, lo que sumaría casos adicionales a sumar en las validaciones si implementáramos completo lo referido a la memoria.

¿Por qué pusimos un guión bajo en `_mergesort()`? Porque lo esperable de una función de ordenamiento es que ordene sobre el mismo vector que recibe y esta implementación está devolviendo un vector nuevo. Lo razonable sería implementar un wrapper:

```
// Ordena el vector v por el algoritmo de mergesort
void mergesort(int v[], size_t n) {
    int *r = _mergesort(v, n);
    memcpy(r, v, n * sizeof(int));
    free(r);
}
```

De la implementación se hace evidente que el método no es in-place.

Sin pedidos de memoria

Sin cambiar la esencia del algoritmo se pueden reimplementar las funciones anteriores para que las mismas reciban un vector auxiliar para operar los resultados. En este caso la idea es que los resultados se vayan ya escribiendo en la posición definitiva dentro del vector. Esto implica que en vez de realizar una recursión clásica pasándole subvectores a las funciones, lo que vamos

a hacer es pasarle los índices de inicio y fin de los mismos. Además las funciones van a conocer que un subvector es la partición al medio de un vector consecutivo.

El algoritmo de merge entonces será:

```
1  // Une las mitades v[desde]..v[medio] y v[medio]..v[desde] y
   ↪ guarda el resultado en v[desde]..v[desde].
2  // Utiliza a aux como vector auxiliar en el proceso.
3  void merge(int v[], size_t desde, size_t medio, size_t hasta, int
   ↪ aux[]) {
4      size_t na = medio - desde;
5      size_t nb = hasta - medio;
6
7      size_t ia = desde;
8      size_t ib = medio + 1;
9      size_t ir = desde;
10
11     while(ia < na && ib < nb)
12         if(v[ia] < v[ib])
13             aux[ir++] = v[ia++];
14         else
15             aux[ir++] = v[ib++];
16
17     while(ia < na)
18         aux[ir++] = v[ia++];
19     while(ib < nb)
20         aux[ir++] = v[ib++];
21
22     for(ir = desde; ir < hasta; ir++)
23         v[ir] = aux[ir];
24 }
```

En esta implementación unimos sobre el vector auxiliar y al final copiamos el vector ordenado sobre su ubicación dentro de la memoria original.

Luego el mergesort se implementará:

```
1  void _mergesort(int v[], size_t desde, size_t hasta, int aux[]) {
2      if(hasta <= desde)
3          return;
4
5      size_t medio = (desde + hasta) / 2;
6
7      _mergesort(v, desde, medio, aux);
8      _mergesort(v, medio + 1, hasta, aux);
9
10     merge(v, desde, medio, hasta, aux);
11 }
12
13 void mergesort(int v[], size_t n) {
14     int *aux = malloc(n * sizeof(int));
15     _mergesort(v, 0, n, aux);
16     free(aux);
17 }
```

Notar que si bien pudimos reducir la cantidad de pedidos de memoria seguimos necesitando hacer un pedido. Esto es porque el método no funciona in-place. Podríamos omitir todos los pedidos de memoria si le pidiéramos a Bárbara que nos pase un buffer de tamaño correspondiente, en ese caso la firma del wrapper podría ser `void mergesort(int v[], size_t n, int aux []).`

22.4.3. Eficiencia

El cómputo de la complejidad temporal del método de mergesort es un poco más complejo que el de selección e inserción dado que se trata de un método recursivo. Al ser recursivo el tiempo de una instancia va a depender de lo que dependan sus subinstancias por lo que nos quedará una ecuación de recurrencia que tendremos que resolver.

Empecemos llamando $T(n)$ al tiempo de llamar a mergesort con un vector de tamaño n. Recordemos la estructura general del algoritmo: Si es el caso base retorno el mismo vector recibido. Si no llamo a mergesort con las dos mitades del vector y luego hago el merge entre esas dos mitades.

Entonces para el caso general $T(n) = 2T(n/2) + an$ dado que tengo dos veces el tiempo que lleve llamar a mergesort con vectores de la mitad del tamaño ($T(n/2)$) y luego la llamada a merge, que para dos vectores de tamaño $n/2$ tiene complejidad $\mathcal{O}(n)$, por lo que habrá una cantidad de tiempo a multiplicando a la cantidad n. Esta es una ecuación de recurrencia, está definida en términos de sí misma. Y, como vimos cuando aprendimos recursividad, tiene que haber un caso particular que termine la recurrencia si no será infinita. Sabemos que $T(0) = T(1) = b$, porque cuando llegamos a un caso base hacemos una operación que ya no depende del tamaño genérico n.

Entonces tenemos que resolver la recurrencia:

$$T(n) = \begin{cases} 2T(n/2) + an & \text{si } n > 1, \\ b & \text{si } n \leq 1. \end{cases}$$

Vamos a empezar haciendo la misma asunción que hicimos cuando resolvimos la complejidad de la búsqueda binaria (sección 17.3): Supongamos que $n = 2^k$, otra vez, una cantidad que podemos dividir por 2 muchas veces sin que aparezcan cantidades impares.

Entonces en el primer paso de nuestra recurrencia tenemos $T(n) = T\left(2^k\right) = 2T\left(2^{k-1}\right) + a2^k$. Ahora podemos desarrollar $T\left(2^{k-1}\right)$ según lo que tenemos en la recurrencia, entonces $T(n) = 2\left(2T\left(2^{k-2}\right) + a2^{k-1}\right) + a2^k = 2^2T\left(2^{k-2}\right) + 2a2^k$.

Podemos seguir haciendo esto k veces, en el paso k obtendremos $T(n) = 2^kT\left(2^{k-k}\right) + ka2^k$. Ahora bien como $2^{k-k} = 2^0 = 1$ entonces $T\left(2^{k-k}\right) = T(1) = b$, entonces nuestra ecuación queda $T(n) = 2^kb + ka2^k$. ¿De dónde salió k?, salió de que asumimos que $n = 2^k \implies k = \log_2 n$. Entonces podemos hacer ese último reemplazo para obtener la ecuación en términos de n:

$$T(n) = nb + \log_2(n)an.$$

En esta ecuación tenemos un término lineal (nb) y un término que es un producto entre n y el logaritmo de n, este último término es mayor, por lo que si aplicamos notación $\mathcal{O}$ podemos descartar el término lineal y la constante a obteniendo entonces que $T(n) = \mathcal{O}(n \log n)$.

Más allá del desarrollo analítico podemos visualizar el resultado de forma gráfica. Empezamos con un problema de tamaño n. Cada vez que lo subdividimos tendremos dos problemas de tamaño $n/2$, que a su vez dividirán en 4 problemas de tamaño $n/4$ y luego en 8 de tamaño $n/8$ hasta que no se pueda partir más. La cantidad de veces que podremos partir será $\log_2 n$, así que terminaremos después de esa cantidad de divisiones. A su vez, el problema de tamaño n se

resolverá en $\mathcal{O}(n)$. Del mismo modo tenemos dos problemas de tamaño $n/2$ que sumados nos vuelven a dar $\mathcal{O}(n)$ y luego tenemos 4 de tamaño $n/4$ donde obtendremos lo mismo. Es decir, en cada subdivisión vamos a operar con particiones del vector que siempre suman n elementos y podemos particionar $\log_2 n$, lo cual implica una solución en tiempo $\mathcal{O}(n \log n)$.

Entonces nuestro algoritmo de mergesort será $\mathcal{O}(n \log n)$ en tiempo y $\mathcal{O}(n)$ en memoria. Si recordamos del inicio de este capítulo, se podía demostrar que ese orden temporal es el mejor posible para algoritmos comparativos, por lo que no vamos a encontrar ningún algoritmo que pueda superar a mergesort para el caso general.

Puede verse que, por ejemplo al igual que en selección, al algoritmo de mergesort no le importa el contenido del vector, es decir, parte al medio tantas veces como puede y luego va efectuando la operación de merge. Por lo que no hay un mejor y un peor caso de este algoritmo, siempre vamos a obtener complejidad $\mathcal{O}(n \log n)$.

22.4.4. Merge más allá de mergesort

Si bien el disparador para enseñar el algoritmo de merge en este curso es que es lo que permite implementar el ordenamiento mergesort, la realidad es que merge por sí solo es un algoritmo muy útil. El poder de merge se da en que se puede sintetizar el contenido de dos vectores diferentes en apenas una pasada por los elementos de ambos, por lo que permite resolver de forma lineal operaciones que a priori no lo parecen.

Por dar un ejemplo, supongamos que tenemos dos conjuntos de elementos A y B, por sencillez de tipo entero. Cada conjunto está representado en un vector y los elementos de dicho vector están ordenados. Ahora bien queremos encontrar el conjunto intersección $R = A \cap B$.

Una aproximación ingenua a este problema caería en la definición de intersección de conjuntos, un elemento x estará en el conjunto intersección sólo si $x \in A \land x \in B$. Si quisiéramos operar esto podríamos verificar si cada uno de los elementos a_i de A pertenece al conjunto B, entonces tendríamos algo así como:

```
1  // Devuelve a intersección b, y la dimensión de este conjunto en
   ↪ nr.
2  int *interseccion(const int a[], size_t na, const int b[], size_t
   ↪ nb, size_t *nr) {
3      // El conjunto ocupará como máximo min(na, nb)
4      int *r = malloc((na < nb ? na : nb) * sizeof(int));
5      if(r == NULL) return NULL;
6
7      *nr = 0;
8
9      for(size_t i = 0; i < na; i++)
10         if(pertenece(b, nb, a[i]))
11             r[(*nr)++] = a[i];
12
13     // Podríamos redimensionar r a *nr con realloc si quisiéramos.
14
15     return r;
16 }
```

Debería verse que iteramos n_a veces haciendo una llamada a la función que nos dice si $a_i \in B$.

¿Cuál es el orden de complejidad de determinar si un elemento está en B? Si no supiéramos nada sobre B deberíamos hacer una búsqueda lineal que será $\mathcal{O}(n_b)$. Pero empezamos planteando el problema diciendo que ambos vectores estaban ordenados por lo que podemos hacer una búsqueda binaria que podemos resolver en $\mathcal{O}(\log n_b)$.

El orden del algoritmo de intersección será entonces $\mathcal{O}(n_a \log n_b)$ donde si asumimos que ambos vectores tienen tamaños comparables podemos decir que es $\mathcal{O}(n \log n)$, nada mal.[2]

Ahora bien si estamos hablando del problema de la intersección de vectores ordenados en una sección que se llama "merge más allá del mergesort" será porque merge tiene algo que aportarnos al respecto de este problema.

Razonemos un poco, si los elementos de A y los de B se encuentran ordenados hay dos casos a considerar sobre ese primer elemento. Si tenemos que $a_0 = b_0$ esto quiere decir que ese elemento está en ambos conjuntos y por lo tanto tiene que formar parte del conjunto intersección. En cambio, si son diferentes uno de ellos tiene que ser menor que el otro. El que sea más chico no puede **nunca** estar en el otro conjunto porque si no estaría primero, entonces ese elemento puede descartarse dado que no va a ser parte del conjunto intersección. Podemos aplicar esta lógica hasta que se agoten los elementos de uno de los dos conjuntos. Cuando se agoten los elementos de un conjunto ya no importan los que queden en el otro, no hay forma que sean parte del conjunto intersección.

Juntando estas ideas podemos programar:

```
1  int *interseccion(const a[], size_t na, const b[], size_t nb,
       ↪ size_t *nr) {
2      int *r = malloc((na < nb ? na : nb) * sizeof(int));
3      if(r == NULL) return NULL;
4
5      *nr = 0;
6      size_t ia = 0, ib = 0;
7      while(ia < na && ib < nb) {
8          if(a[ia] == b[ib]) {
9              r[(*nr)++] = a[ia++];    // Da igual si tomamos este o
                   ↪ b[ib]: son iguales.
10             ib++;                    // ¿Qué pasaría si no
                   ↪ incrementamos ib?: Nada. Pensalo
11         }
12         else if(a[ia] < b[ib])
13             ia++;
14         else
15             ib++;
16     }
17
18     return r;
19 }
```

No es de sorprendernos que si planteamos la idea de merge entonces el algoritmo de intersección nos haya quedado como un único recorrido sobre los dos vectores, el orden de complejidad será entonces $\mathcal{O}(n_a + n_b)$, mejor que el resultado que habíamos obtenido para la implementación ingenua.

Ahora bien, podríamos decir que sólo podemos aplicar el algoritmo de merge si los vectores están previamente ordenados, es cierto. Pero si decimos eso tenemos que ver que si los vectores no estuvieran ordenados tampoco podríamos hacer la búsqueda binaria en el algoritmo ingenuo y obtendríamos una complejidad $\mathcal{O}(n^2)$, totalmente inaceptable. Si los vectores estuvieran desordenados entonces sería más eficiente primero ordenarlos por un método como mergesort y después computar su intersección, ahora sí, con cualquiera de los dos métodos, total ya no podremos mejorar el $\mathcal{O}(n \log n)$ que gastamos en el ordenamiento.

[2] Y más que decir que tienen tamaños comparables, nos convendría fijarnos cuál de los dos vectores es el más corto e interar sobre los elementos de ese que son menos.

No lo mencionamos previamente pero si representamos los conjuntos mediante vectores ordenados deberíamos garantizar que el resultado R también esté ordenado y puede verse que ambos métodos que propusimos lo verifican. También tendría sentido que no hubiera elementos repetidos en ninguno de los vectores, cosa que si se cumple en A y A se va a cumplir también en R para ambos métodos.

Esta implementación de la intersección que planteamos utilizando la lógica de merge puede generalizarse para otras operaciones de pertenencia. En todos los casos podemos establecer primero una cota de la cantidad máxima de elementos va a tener el resultado final, y en todos los casos podremos decidir qué hacer según la relación entre los primeros elementos y qué hacer con el resto del vector al terminar la iteración principal. Sin ir más lejos la implementación original de merge se parece mucho a la unión entre dos conjuntos, con el detalle de que si hay elementos en ambos conjuntos estarán repetidos en el resultado, ¿cómo se modificaría la implementación para no incluirlos? Otras operaciones que podemos hacer con la lógica del merge es, por ejemplo, encontrar todos los elementos de A que no estén en B, etc.

22.5. Quicksort

Llegamos al último método que vamos a presentar en este curso, el método de quicksort. Este, al igual que mergesort, también es un método de ordenamiento recursivo.

La idea del quicksort consiste en empezar eligiendo un elemento arbitrario del vector (el primero, el último, el del medio, alguno, ya lo discutiremos más adelante) denominado pivote. El pivote se utilizará como referencia para construir dos subvectores: Uno con todos los elementos menores al pivote y otro con todos los elementos mayores al pivote. Luego de forma recursiva se ordenarán ambos subvectores. A diferencia del mergesort donde la vuelta de la llamada recursiva no era evidente si ahora tenemos un subvector ordenado con todos los elementos menores al pivote y otro subvector con todos los elementos mayores también ordenado es evidente que el vector resultante tendrá que ser la concatenación de los elementos menores, el pivote y los elementos mayores.

Entonces la implementación será:

```
1  void quicksort(int v[], size_t n) {
2      if(n <= 1)
3          return;
4
5      // Elección de un pivote:
6
7      size_t npivote = n / 2;  // Elegimos de pivote el del medio,
           ↪ es arbitario.
8      int pivote = v[npivote];
9
10     // Partición en elementos menores y mayores:
11
12     int *menores = malloc((n - 1) * sizeof(int));
13     int *mayores = malloc((n - 1) * sizeof(int));
14
15     size_t nmenores = 0; nmayores = 0;
16     for(size_t i = 0; i < n; i++) {
17         if(i == npivote)
18             continue;
19
20         if(v[i] < pivote)
21             menores[nmenores++] = v[i];
```

```
22          else
23              mayores[nmayores++] = v[i];
24      }
25
26      // Ordenamos recursivamente menores y mayores:
27
28      quicksort(menores, nmenores);
29      quicksort(mayores, nmayores);
30
31      // Concatenamos menores + pivote + mayores:
32
33      memcpy(v, menores, nmenores * sizeof(int));
34      v[nmenores] = pivote;
35      memcpy(v + nmenores + 1, mayores, nmayores * sizeof(int));
36
37      free(menores);
38      free(mayores);
39  }
```

Esta implementación es literalmente el algoritmo que se explicó. Para estar completa habría que agregar las validaciones de memoria que se omitieron.

Como se puede ver, estamos teniendo que consumir $n-1$ de memoria en cada llamada para los subvectores de elementos menores y mayores. Nuestra implementación no es in-place.

22.5.1. Eficiencia

Antes de analizar la complejidad del algoritmo intentemos entender la idea de la reducción del problema. En cada llamada recursiva estamos sacando un elemento del vector, que es el pivote, por lo que los subvectores que ordenamos recursivamente tienen un elemento menos. Pero del capítulo de recursividad nos debería haber quedado en claro que sacar un elemento de un vector no conlleva de por sí a una buena solución recursiva.

La idea fuerte de reducción del quicksort está en que si tenemos un buen criterio para elegir el pivote esperaríamos que la cantidad de elementos menores a él sea similar a la cantidad de elementos mayores a él. Es decir, reduzcamos el problema en dos mitades.

Esto que parece ser un objetivo simple no es fácil de garantizar en la práctica partiendo de la base de que tenemos un vector de n elementos desordenados y no podemos dedicar un esfuerzo computacional a encontrar el pivote tal que sea la mediana del conjunto.

Para entender cómo afecta el pivote pensemos algunos casos extremos. Supongamos que elegimos como pivote a v_0, el primer elemento del vector. Si el vector estuviera ordenado, ¿qué pasaría? Es evidente ver que si el vector está ordenado entonces v_0 es el menor elemento del vector, por lo tanto todos los demás elementos serán mayores. Entonces en vez de reducir el problema en dos subproblemas de tamaño $\frac{n-1}{2}$ resulta que resolvimos el problema en un subproblema de tamaño 0 y otro subproblema de tamaño $n-1$ y como el vector está ordenado a cada llamada recursiva va a pasarle lo mismo. Si cada llamada recursiva reduce el problema en un elemento y para armar el subvector necesita iterar todos los elementos la complejidad de cada llamada individual será $\mathcal{O}(n)$ y habrá n llamadas por lo que estaremos ante un caso $\mathcal{O}(n^2)$. Esto mismo se repetirá si eligiéramos como pivote el último elemento o si el vector está ordenado al revés. Siempre que no queden balanceadas las dos mitades estaremos en complejidades cuadráticas.

El problema es dependiente de dos cosas, una es de la elección del pivote y la otra es de cuál es el orden de los datos. Podés pensar que una implementación como la que hicimos, donde tomamos de pivote al elemento del medio resuelve el problema. Si bien es cierto que en el

caso de que el vector esté ordenado tomar el elemento central como pivote va a garantizar que los subvectores sean justo de la mitad del tamaño, esto no necesariamente sea así para el caso genérico de un vector desordenado, que a fin de cuentas será el tipo de vectores que querremos ordenar. Hay múltiples técnicas para elegir el mejor pivote de diverso grado de complejidad y no vamos a profundizar en ninguna de ellas en este curso.

Volviendo al tema de la eficiencia, el peor caso es cuadrático, ¿pero cuánto será el mejor caso?

Si cada vez que partimos en menores y mayores cada uno de estos tiene la mitad (menos uno) del problema original, y si el pre y postprocesamiento de los datos es lineal, entonces vamos a estar ante la misma ecuación de recurrencia del mergesort y sabemos que eso resulta en complejidad $\mathcal{O}(n \log n)$.

No perdamos de vista que, incluso con la mejor elección del pivote, siempre podemos recibir un vector ordenado de una forma patológica tal que las particiones terminen desbalanceadas y quedemos del lado cuadrático. Así y todo, podemos decir que en promedio el quicksort se comporta $\mathcal{O}(n \log n)$.

22.5.2. In-place

Si bien cuando presentamos el algoritmo dijimos que nuestra implementación no era in-place eso no significa que no pueda implementarse una que lo sea. Presentamos la versión que utiliza vectores auxiliares porque es la implementación natural que surge de la descripción del problema. Es decir es una solución intuitiva para el algoritmo propuesto.

La idea de la implementación in-place es lograr realizar la separación entre los elementos menores y los mayores sobre el mismo vector, sin utilizar memoria auxiliar.

¿Cómo podemos lograr esto? La idea es pensar en el vector como si tuviera tres secciones: Al principio del vector los elementos menores al pivote, al final del vector los elementos más grandes y en el medio del vector los elementos que todavía no clasificamos entre menores y mayores. Cuando el algoritmo comienza todos los vectores pertenecen a la última categoría, no sabemos qué son. A medida que vamos avanzando con la clasificación iremos dejando al comienzo los elementos menores y al final los elementos mayores, hasta que clasifiquemos a todos.

La idea es así:

```
1  // Divide a v en dos subvectores, devuelve la posición del pivote.
2  size_t clasificar(int v[], size_t n) {
3      size_t npivote = n / 2;      // Arbitrariamente elegimos el del
       ↪  medio, otra vez.
4      int pivote = v[npivote];
5
6      // Guardamos el pivote al final temporariamente:
7      swap(v + npivote, v + n - 1);
8
9      size_t imenores = 0;         // Los menores estarán entre 0..
       ↪ imenores
10     size_t imayores = n - 2;     // Los mayores estarán entre
       ↪ imayores..n-2.
11     while(imenores < imayores) {
12         if(v[imenores] < pivote)
13             // Si el primer elemento después del bloque de menores
                  ↪  es menor que el pivote, ampliamos el bloque de
                  ↪ menores:
14             imenores++;
```

```
15          else {
16              // En cambio si no es menor lo mandamos al bloque de
                ↪ mayores:
17              swap(v + imenores, v + imayores);
18              imayores--;
19          }
20      }
21
22      // Al terminar la iteración todos los menores están al
            ↪ comienzo, le siguen los mayores y en la última posición
            ↪ está el pivote.
23      // La iteración termina cuando imenores == imayores.
24
25      // Pero el pivote tiene que estar justo después de los menores
            ↪ :
26      swap(v + imenores, v + n - 1);
27      return imenores;
28  }
```

Más allá de la implementación y los detalles, ¿te das cuenta qué tiene de problemática la idea de clasificar de esta manera sobre el mismo vector intercambiando elementos del principio con el final? Es la única propiedad de clasificación de algoritmos de ordenamiento que mencionamos en todos los demás métodos pero omitimos cuando hablamos del quicksort: Desordena el orden relativo de los elementos originales. Entonces el quicksort implementado sobre esta función de clasificación no será estable[3].

Teniendo implementada esta clasificación que nos divide a v en los dos subvectores entonces podemos implementar el método de ordenamiento:

```
1  void quicksort(int v[], size_t n) {
2      if(n <= 1)
3          return;
4
5      size_t npivote = clasificar(v, n);
6
7      quicksort(v, npivote - 1);                  // Ordenamos los
           ↪ menores al pivote
8      quicksort(v + npivote + 1, n - npivote);    // Ordenamos los
           ↪ mayores al pivote.
9  }
```

¿Falta algo al final? No, no falta nada. Como la función de clasificación nos dejó los subvectores en su posición relativa al pivote la llamada recursiva trabaja en el lugar definitivo. Literalmente eso es lo que significa in-place.

Esta nueva versión presentada tiene una complejidad espacial $\mathcal{O}(1)$, en contrapartida perdimos la estabilidad.

22.6. Resumen

Hagamos una comparativa de los 4 métodos que presentamos:

[3] A diferencia del quicksort que implementamos antes que preservaba los órdenes.

Método	Mejor caso	Caso promedio	Peor caso	In-place	Estable
Selección	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	Sí	Sí
Inserción	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	Sí	Sí
Mergesort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	No	Sí
Quicksort (I)	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	No	Sí
Quicksort (II)	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	Sí	No

No vamos a volver a mencionar los puntos fuertes y débiles de cada uno porque para eso ya está el capítulo. De todos modos ahora que tenemos todos los métodos juntos para compararlos, ¿considerás que hay uno que sea mejor que los otros tres? Tomate un tiempo para pensar la respuesta.

En la sección 8.11.1 vimos que la `<stdlib.h>` ya trae programado un método para ordenar arreglos. Esta función se llama `qsort()` que no es otra cosa que una abreviatura de quicksort. ¿Es el método que elegiste en el párrafo anterior?, ¿se te ocurre cuál fue el criterio para haber elegido ese método?

Probablemente esta sea una de las mejores ideas para cerrar este texto. En buena parte de la ingeniería no existen soluciones universales si no que siempre estamos ante soluciones de compromiso. Sí existen soluciones que no aportan nada bueno, pero generalmente no hay una solución ganadora que cubra todos los aspectos. En este caso a la hora de implementar una función de biblioteca ganó la robustez de una solución que no necesita utilizar memoria dinámica, incluso sacrificando estabilidad y corriendo el riesgo de degenerar a órdenes de complejidad cuadráticos. Una función de biblioteca que intente ordenar un vector y requiera duplicar la memoria o, peor aún, falle es inaceptable. Es incluso inaceptable contrastada con el riesgo de en algunos casos tardar tiempos absurdos. No hay una sola forma de resolver los problemas, hay soluciones para distintos contextos y el arte es desarrollar el criterio que permita elegir la mejor solución para el que nos toca.