

Guía del Piscinero Galáctico

josesanc

10/04/2023

Índice

Introducción	8
Conceptos básicos de C	9
Operadores	10
Operadores a nivel de bits	11
Comparadores	13
Leyes de De Morgan	13
Cortocircuito	13
Tipos	14
Int	14
Char	14
Float y Double	14
Short y Long	14
Unsigned	14
size_t	14
Void	14
Bool	15
Sizeof	15
Casting	15
Estructuras de control	15
Condicionales	15
Bucles	16
Otras	17
Funciones	18
Prototipo	18
Llamada	18
Declaración	18
Return	18
Static	18
Main	18
Punteros	19
Punteros a funciones	19
Arrays	19

Cadenas	19
Structs.....	20
Uniones	20
Enumeraciones.....	20
Constantes.....	20
Técnicas de programación.....	21
Recursividad.....	22
Búsqueda.....	23
Búsqueda binaria	23
Ordenación	24
Burbuja - $O(n^2)$	24
Inserción - $O(n^2)$	24
Mergesort - $O(n \cdot \log n)$	25
Quicksort - $O(n \cdot \log n)$	25
Fuerza bruta	25
Backtracking (marcha atrás).....	25
Programación Dinámica	26
Algoritmos Voraces (Greedy).....	26
Divide y Vencerás.....	26
Librerías.....	27
Unistd.....	28
Write	28
Stdio.....	28
Printf	28
Stdlib	28
Malloc	28
Free.....	29
Atoi.....	29
NULL	29
String.....	29
Strcpy.....	29
Strlen	29
Strcat.....	29

Strdup.....	29
Erno.....	30
Memoria Persistente. Archivos	30
Open	30
Close	30
Read	30
Write	30
Compilación	31
Makefiles	32
Reglas generales	33
Compilando con dependencias	33
Listas.....	35
Pilas (Stack).....	36
Colas (Queue)	36
Árboles (binarios)	37
Diccionarios.....	38
Misceláneo	39
Errores	40
Bus error y Segmentation Fault.....	40
Overflow.....	40
Directivas de preprocesador	41
Include.....	41
Define	41
Condicionales.....	41
Cursus	42
Señales	43
Kill	43
PID	43
signal y sigaction	43
MLX (minilibx)	43
Concurrencia.....	44
Redirección y file descriptors.....	45
Pipes.....	45

Matemáticas	46
Números complejos	46
Trigonometría	46
Anexo	47
Bits, bytes y hexadecimal	48
¿Cómo interpretar el binario?	48
¿Cómo interpretar el hexadecimal?	48
Escribiendo hexadecimal y octal en C.....	48
Formatos de bases	48
Control de fuga de memoria.....	49
Notaciones.....	49
snake_case	49
camelCase.....	49
Comentarios	50
STD.....	50
STDIN - 0	50
STDOUT - 1	50
STDERR - 2.....	50
La gran O - Big O Notation	50
Usando Git para el control de versiones.....	51
Commits.....	51
Ramas.....	51
Merge	51
Ramas remotas	51
Forks	52
Git ignore	52
Vi Machine (Vim).....	52
Consejos.....	53
Doxygen Documentation	54
Legibilidad del código	54
Directivas del código.....	54
lldb	55
Evitando relink.....	55

AVISO

Este documento está basado en mi experiencia de programación mediante 42, universitaria e Internet.

Está realizado por pasión, por lo que sobra decir que no está pensado para comercializarse ni nada de eso.

Existen, además, secciones que no corresponden a contenidos de la piscina, por lo que pueden ser obviadas.

No obstante, a cualquier estudiante que le pueda servir, se lo podéis pasar; a piscineros, no.

Con cariño, la primera página que nadie lee.

SUGERENCIAS

Puesto que este documento pretende ser actualizado, se desaconseja su impresión en papel. Este no es un mensaje de salvar el planeta sino de no dejar tu copia con posibles errores o peores explicaciones.



En caso de encontrar una errata en el documento, puedes dejar un comentario constructivo en la versión publicada en Google Drive o contactar con el autor para mejorarlo.



Introducción

Digamos que no has visto nada de programación hasta la fecha, las aplicaciones y programas te parecen magia y casi que tienes la razón en ello. La capacidad de cómputo ha ido y sigue creciendo a niveles que nadie se imaginaría década atrás cuando usaban válvulas de vacío o cintas magnéticas para usar los ordenadores.

En aquellos tiempos se propuso el proyecto MULTICS un sistema operativo tan ambicioso, del cual surgiría UNICS, que fue reescrito en C y renombrado a UNIX más tarde; de este sistema operativo surgirán algo después GNU/Linux y MacOS.


Pero, ¿por qué C? Hasta entonces ya habían aparecido lenguajes como Fortran y Cobol; sin embargo, C surge como un lenguaje de programación de alto nivel (abstracción) con capacidad de controlar la memoria a bajo nivel.

Los niveles de programación van desde el lenguaje de ensamblador, instrucciones al ordenador que se traducen a binario para ser ejecutadas por el procesador; al alto nivel donde encontramos lenguajes como C, Java, Python, JavaScript... Que bien pasan por un proceso de compilación como C (o pseudo-compilación como Java) o de interpretación como Python y JavaScript.

Normalmente se destaca en estos últimos la ventaja de ser más portables frente a los compilados como C, puesto que determinadas instrucciones ya compiladas son exclusivas de sistemas operativos y se requeriría recompilar el programa para otro.

Por suerte, programar hoy en día la programación es menos duro, pero sus distintas ramas pueden abrumar. C y C++ siguen siendo grandes aliados y enemigos de programadores centrados en la rapidez de sus programas en servidores, sistemas operativos y embebidos.

Como dijo Mecano en los 80: Qué pesado, siempre pensando en el pasado...



Conceptos básicos de C

Operadores

Suma y resta

Son operadores tan sencillos que no deberían tener un apartado, o tal vez sí... Para sumar constantes o valores de dos variables `var1` y `var2` usamos:

`var1 + var2`

Mientras que los restamos con la siguiente expresión:

`var1 - var2`

Por sí solas, estas expresiones no actualizan el valor de ninguna de las variables, para ello será necesario la asignación o alguna de las siguientes expresiones:

`var1++`, `++var1`, `var1--`, `--var1`

Que aumentan o decrementan en uno, respectivamente y asignan ese nuevo valor a la misma variable.

Curiosidad

La evolución de C, C++, obtiene su nombre de tales expresiones

El orden del signo puede importar según cuándo se use. En el caso de que el **signo** se encuentre **antes** del nombre de la variable, la **asignación** se realizará **antes** de usar su valor; mientras que ponerlo detrás hará que se actualice tras usarlo.

Multipliación

Para multiplicar constantes o los valores de dos variables `var1` y `var2` usamos:

`var1 * var2`

Curiosidad

Una multiplicación de con un múltiplo de 2 puede convertirse a otra instrucción de menos costosa que la multiplicación, el desplazado de bits.

División

Para dividir constantes o los valores de dos variables `var1` y `var2` usamos:

`var1 / var2`

La división entre dos enteros dará como resultado un entero, el **cociente** sin decimales; para obtener los decimales, bien el dividendo o divisor deberán ser tipos de punto flotante (double, float o constante con decimales).

Módulo

Para obtener el resto de la división de constantes o los valores de dos variables `var1` y `var2` usamos:

`var1 % var2`

Es una buena herramienta para saber si un número es par, divisible entre otro o incluso hacer recorridos circulares (reiniciar el índice cuando llegue al máximo).

Asignación

Cuando queremos dar un nuevo valor a una variable existente o recién declarada usamos la asignación, =

```
var1 = var2 + var3;
```

La variable se actualiza tras calcular el resultado, por lo que es posible asignar con la misma variable:

```
var1 = var1 + var2;
```

, por suerte no funciona como una ecuación matemática; en cuyo caso estaríamos acabados.

Esta última asignación se puede sustituir por la siguiente:

```
var1 += var2;
```

, que indica que se usará la misma variable en la asignación. -=, *=, /= o %= realizarían la misma función para sus operadores con el orden seguido anteriormente.

Curiosidad

En C es posible asignar más de una variable en una misma línea. Además, la asignación devuelve un valor, por lo que sería posible aprovecharla para un condicional.

Operadores a nivel de bits

Si no estás familiarizado con los bits, puede resultar de utilidad el [apartado del anexo](#).

AND

Para operar con AND los bits de constantes o los valores de dos variables var1 y var2 usamos:

```
x & y
```

Con cada uno de sus bits se seguirá la tabla de verdad de AND:

x	y	res
0	0	0
0	1	0
1	0	0
1	1	1

Dos bits operados con AND darán como resultado 1 si ambos son 1.

OR

Para operar con OR los bits de constantes o los valores de dos variables var1 y var2 usamos:

$$x \mid y$$

Con cada uno de sus bits se seguirá la tabla de verdad de OR:

x	y	res
0	0	0
0	1	1
1	0	1
1	1	1

Dos bits operados con OR darán como resultado 1 si ambos no son 0.

(E)XOR

Para operar con XOR los bits de constantes o los valores de dos variables var1 y var2 usamos:

$$x \wedge y$$

Con cada uno de sus bits se seguirá la tabla de verdad de XOR:

x	y	res
0	0	0
0	1	1
1	0	1
1	1	0

¡Cuidado!

Aunque en el lenguaje matemático se refiere la potencia, no existe un operador de la potencia.

Su lógica es parecida a la suma de bits sin acarreo, es decir, “sin llevarse una”. Dos bits operados con XOR darán como resultado 1 si no son iguales.

Desplazamiento

Para desplazar n bits hacia la izquierda (<<) o derecha (>>) un valor usamos:

$$x \ll y$$

$$x \gg y$$

, respectivamente. Por ejemplo:

$$0 \dots 1000 \gg 1 = 0 \dots 0100;$$

$$0 \dots 1000 \gg 3 = 0 \dots 0001;$$

$$0 \dots 1000 \gg 2 = 0 \dots 0010;$$

$$0 \dots 1000 \gg 4 = 0 \dots 0000;$$

Comparadores

<code>==</code>	Igual
<code>!=</code>	Distinto
<code>!</code>	No (NOT)
<code>&&</code>	Y (AND)
<code> </code>	O (OR)
<code>></code>	Mayor
<code>>=</code>	Mayor o igual
<code><=</code>	Menor o igual
<code><</code>	Menor

El resultado de las condiciones será 0 en caso de falso o 1, verdadero.

Leyes de De Morgan

Sí, llevas dos “de”, una de ellas es su apellido. Aun no siendo exclusivo de C, las leyes de lógica booleana pueden ayudarte a tener un código más limpio y/o funcional. En C, las leyes se podrían resumir en las siguientes igualdades:

$$\begin{array}{ccc} \neg(c1 \ \&\& \ c2) & \rightarrow & \neg c1 \ || \ \neg c2 \\ & \text{y} & \\ \neg(c1 \ || \ c2) & \rightarrow & \neg c1 \ \&\& \ \neg c2 \end{array}$$

Cortocircuito

En C, como en otros lenguajes, las condiciones se pueden ejecutar en cortocircuito, es decir, no se comprueban todas las condiciones si una es excluyente o hace innecesaria la comprobación de la otra. Veámoslo con dos ejemplos:

`c1 || c2`

, en este caso, la primera condición se cumple y la segunda no; no obstante, esta última no llegará a comprobarse, puesto que ya se puede concluir que es verdadero, según la tabla de verdad de OR.

`c1 && c2`

, contrario al OR, en las condiciones AND, con que la primera resulte ser falsa, se descartará comprobar el resto.

La evaluación en cortocircuito sigue el orden en el que se escriben, por lo que puede convenir dejar condiciones más livianas de procesado primeras para descartar otras y hacer nuestro programa más eficiente.

Tipos

Int

Los tipos int siguen el formato de complemento a 2 con 31 bits reservados para expresar el número más uno para el signo (4 bytes en total). Sus valores oscilan desde el -2^{31} al $2^{31} - 1$ sin decimales, por lo que no es posible expresar el 2^{31} con enteros con signos.

Char

El tipo char sigue también el formato de complemento a 2 sobre un byte (8 bits) de tamaño. Sus valores oscilan entre el -128 y 127. Si bien es posible asignar valores de caracteres como números enteros, resulta mejor escribir un carácter entre comillas simples para entender a simple vista los valores que se asignan. Por ejemplo:

char x = 48;

char x = '0';

Los caracteres especiales deben estar precedidos de la barra de escape, '\'. Para conversión de unidades es posible sumar '0' al número.

La tabla ASCII

El ASCII es un estándar para codificar caracteres en bits. Originalmente se reservó uno de los 8 bits como código de paridad (control de errores en las comunicaciones), aunque un par de décadas más tardes, se estandarizó también el ASCII extendido, que va desde el carácter 0 al 255, para interpretarlos en C siguiendo el orden de la tabla extendida, el carácter debería ir sin signo (unsigned).

Float y Double

A diferencia de los anteriores, se basa en el estándar IEEE 754 para expresar binario con decimales, son los tipos de punto flotante. En el caso del tipo float se dispone de 4 bytes, mientras que los double ocupan 8.

Short y Long

Funcionan al igual que los anteriores con menos o más bits reservados para almacenar los valores en memoria, respectivamente.

Unsigned

Si estamos interesados en asegurar que el número no pueda ser menor a 0, negativo, o queremos obtener un rango de valores algo más grande, pueden ser útiles los tipos unsigned, que interpretan el bit que se usaba como indicativo de positivo o negativo para expresar más valores, de 0 a $2^{32} - 1$ en int y 0 a 255 en char, por ejemplo.

size_t

Es una forma alternativa de llamar a unsigned int/long en algunas librerías.

Void

Cuando queremos expresar un tipo indefinido usamos void.

Bool

En la mayoría de los lenguajes de programación actuales se incluye el tipo booleano, una variable que puede almacenar el valor 0, falso o 1, verdadero. Si bien existen librerías que permiten su uso, C no los tiene implementados de base.

Sizeof

Una función básica de C que nos permite saber el tamaño de un tipo es sizeof. Esta recibirá por parámetro un tipo o variable y devolverá el número de bytes que se reservan para guardar sus valores.

¡Atención!

Esta función no devuelve la longitud de una cadena.

Casting

El proceso de casting no es más que la conversión de tipos. En ocasiones podemos cambiar el tipo de dato con el que se está trabajando por otro, por distintos motivos. Para ello, podemos expresarlo, (casting explícito):

```
int x = (int) 4.5;
```

, que convertiría el 4.5 a 4, en este caso. O realizarlo implícito, si el tipo del argumento de la función a la que se llama permite la conversión.

Estructuras de control

Las estructuras de control nos permiten manejar el flujo de ejecución del programa, según las condiciones que se expresen.

Condicionales

If

```
if (conditional)
{
    /* Code */
}
```

Si se quiere especificar un comportamiento de contrario puede usarse **else**, “si no”:

```
if (conditional)
{
    /* Code */
} else {
    /* Code */
}
```

Del mismo modo, si queremos que, tras comprobar una condición, se compruebe otra, se puede usar **else if** “si no, pero si”.

Operador ternario

Cuando un if (normalmente para una asignación) se puede resumir en la siguiente estructura:

```
if (conditional)
    /* Code line 1 */
else
    /* Code line 2 */
```

, es posible usar el operador ternario:

```
(conditional) ? Code line 1 : Code line 2
```

Switch

Como una estructura de control basado en el valor de una variable tenemos también el switch. Su forma es la siguiente

```
switch (variable) {
    case (constante1):
        /* Code */
        break;
    case (constante2):
        /* Code */
        break;
    . . .
    default:
        /* Code */
}
```

Un caso básico, de ejemplo, es devolver el número de días según el mes que indique la variable.

Bucles

While

Se trata de un bucle sobre el que se itera en base a una condición, si esta es constante bien no entrará en este nunca o bien se quedará para siempre.

```
while (conditional) {
    /* Code */
}
```


Break

Consiste en la salida forzada del bucle sin que se compruebe la condición que lo mantiene.

```
break;
```

Continue

Consiste en el salto a la siguiente iteración del bucle comprobando la condición que lo mantiene.

```
continue;
```

Do While

El funcionamiento es casi idéntico al de un while a excepción del orden en el que se comprueba la condición del bucle, al final. Consecuentemente, habrá una iteración (de no haber break o return que lo impida).

```
do {  
    /* Code */  
} while (conditional);
```

For

A la hora de definir cuántas veces queremos que se ejecute un bloque de código o iterar sobre un array, puede ser conveniente el uso de for:

```
for (int i = 0; i < 5; i++) {  
    /* Code */  
}
```

En él se distingue una variable que se inicializa y es accesible hasta que acabe el bucle, una condición que debe cumplirse al principio de cada iteración y una instrucción que se realizará al final de esta.

Curiosidad

Es posible obviar el uso de los corchetes si el código se puede expresar en una línea, aunque puede disminuir la legibilidad del código.

Otras

Goto

Aunque se trata de una mala práctica, recuerda a cómo funcionan los saltos de instrucciones a bajo nivel, permitiendo que la ejecución del programa sea más enrevesada, pero que pudiera explotar alguna capacidad a cambio. Debido a que la mantenibilidad del código es algo bastantepreciado, el uso de goto considerablemente bajo.

Funciones

Una función es un bloque de código (instrucciones) que se puede llamar para que se ejecute. Los **argumentos** son siempre **copias** de los que son pasados en las llamadas, por lo que, para el paso por referencia (uso de punteros) se debe tener en cuenta que, realmente, se pasa una copia de la dirección de memoria, a la que se accederá para modificar el valor al que se apunta. Su formato es el siguiente:

Prototipo

```
tipo nombre(argumentos);
```

Llamada

```
nombre(argumentos);
```

Declaración

```
tipo nombre(argumentos) {  
    /* Code */  
}
```

Return

```
tipo_no_void nombre(argumentos) {  
    /* Code */  
    return (x);  
}
```

Static

La cláusula static sirve para hacer privada la **función**, es decir, solo se podrá llamar desde el archivo desde el que se creó. O bien para conservar el valor entre funciones si se usa frente a una **variable**...

Main

En el proceso de compilación, cuando el linker quiere seguir el orden de ejecución busca la función principal, main, desde la cual compilar el programa. El tipo de la función es int y se deberá devolver 0 en caso de éxito u otro valor (1) de haber error.

La compilación se puede realizar sin main si solo realizamos la conversión a binario (archivos .o), que se puede especificar en los comandos de **cc** o **gcc** con la flag **-c**.

Argumentos

Un programa puede permitir la entrada de argumentos por consola, para ello es necesario que la función main tenga una estructura similar a la siguiente:

```
int main(int argc, char **argv). . .
```

Donde argc es el número de argumentos (contando el propio nombre del programa como argv[0]) y un array de cadenas de argc longitud.

Punteros

Un puntero es una **dirección** de memoria

Declaración	&	Dirección de
	*	Apunta a
Expresión	*	Contenido de la dirección

Un ejemplo, que resume su declaración y uso es el siguiente:

```
int x = 5;
int *ptr = &x;
*ptr = 7;
```

Es similar al paso por referencia en otros lenguajes de programación.

Punteros a funciones

Como un puntero es tan solo una dirección de memoria y la memoria de datos e instrucciones están unidas, es posible apuntar a una función. Por ejemplo:

```
void (*ft_ptr)(char**) = ft_aux;
```

, obteniendo así un puntero a la función `ft_aux` cuyo argumento de entrada es un doble puntero de `char`. El uso de este puntero será similar al de una llamada a una función.

Arrays

Un array es una secuencia de posiciones en memoria reservada para un determinado tipo. Su longitud es fija, por lo que solo se puede acceder a posiciones de 0 al $n - 1$.

```
char str[5]= "Hola";
```

Un array se puede inicializar en la misma línea de la declaración con corchetes (y simplificado con doble comillas en caso de cadena). Realizar la inicialización del array al completo después de la declaración conlleva problemas en la compilación.

```
int digitos[10]= {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

Es equivalente a un puntero constante (con memoria reservada) en C.

Cadenas

Una cadena (string) es un array de caracteres terminado en el carácter nulo. El tamaño de esta, aunque haya más espacio reservado en memoria, se interpretará como la distancia desde el principio de la cadena al carácter nulo, `'\0'`.

Structs

Son variables registros que definir al principio del archivo o en headers (.h). Para acceder a uno de sus campos se usa como notación el punto, '.'. Su uso es lo más parecido a objetos que verás en C si provienes de lenguajes OOP.

```
struct nombre {  
    tipo var1;  
    tipo var2;  
    ...  
};
```

Notaciones

Puesto que, de pasarlo como argumento estaríamos pasando una copia que no actualiza la original, es frecuente pasar los structs en punteros. Para acceder a los valores de sus campos se usaría la siguiente notación,

```
(*nombre).campo1
```

No obstante, para mejorar la legibilidad del código existe una notación alternativa y equivalente a la anterior:

```
nombre -> campo1
```

Uniones

Es parecido a una estructura, pero donde todas las variables comparten la misma memoria. La declaración se da como:

```
union nombre {  
    tipo var1;  
    tipo var2;  
    ...  
};
```

Enumeraciones

Es una lista de valores constantes. Por ejemplo:

```
enum coord {NORTH, SOUTH, WEST, EAST};
```

Constantes

Una constante única puede ser declarada con la palabra clave const:

```
const double pi = 3.14159
```



Técnicas de programación

Recursividad

La recursividad consiste en la llamada dentro de una función a ella misma. Algunas soluciones son más eficientes, sencilla y/o elegantes por medio de la recursividad, aunque también puede ocurrir lo contrario; dependerá del problema que estemos afrontando.

La estructura de la recursividad debe ser la siguiente para que converja:

$$f(x) = \begin{cases} \text{Caso Base} \rightarrow y \\ \text{Caso General} \rightarrow f(x') \end{cases}$$

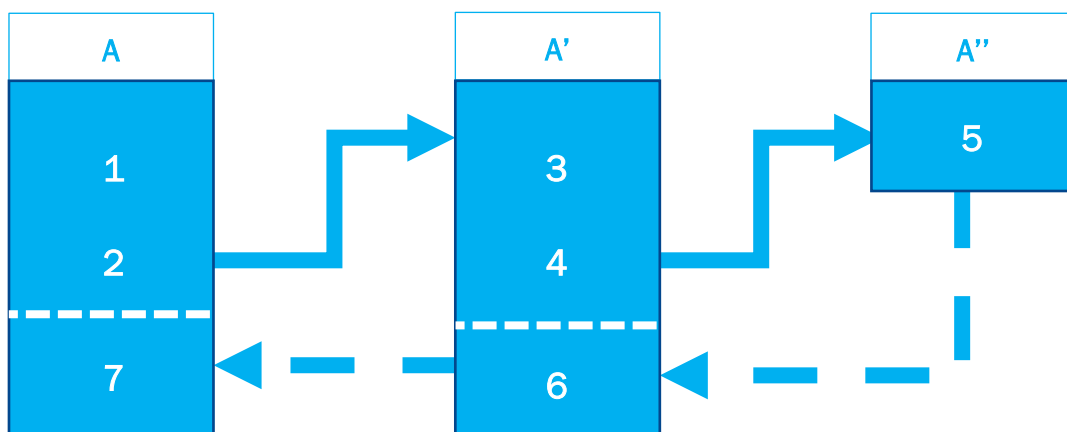
En el caso base se devuelve un resultado y termina así la cadena de llamadas, mientras que en el caso general se llama a la misma función con un argumento distinto.

$$\text{Factorial: } f(x) = \begin{cases} 1 & x = 0 \\ x \cdot f(x - 1) & x > 0 \end{cases}$$

$$\text{Fibonacci: } f(x) = \begin{cases} 0 & x = 0 \\ 1 & x = 1 \\ f(x - 1) + f(x - 2) & x > 1 \end{cases}$$

Las utilidades de la recursividad son:

- **Cálculo**, como en estos dos últimos ejemplos.
- **Retrasar** la ejecución de una instrucción, poniéndola tras la llamada a la función. Esto se debe a que la primera vez que será ejecutada en la última llamada a la función. Es un concepto que puede tardar en hacer “clic”, por lo que aquí un ejemplo más visual:



En otras palabras, conseguiríamos que la tercera instrucción de A se ejecutase después de que la tercera instrucción de A' haya terminado. El orden de ejecución se puede seguir con el orden de los números.

Búsqueda

Supongamos que queremos realizar una búsqueda del 5 en un array de 5 posiciones, las comprobaciones podrían realizarse tal que:

8 3 2 1 5

8 3 2 1 5

8 3 2 1 5

8 3 2 1 5

8 3 2 1 5

Para hacer búsquedas en arrays no ordenados, lo indicado es buscar en cada una de las posiciones, saliendo del bucle una vez encontrada la posición en la que aparece y no seguir explorando el resto de los elementos del array.

Búsqueda binaria

La búsqueda binaria aprovecha que el array está ya ordenado para reducir las comprobaciones necesarias. Si queremos encontrar un número y empezamos a buscar por la mitad del array, podemos descartar la mitad de este según si el elemento intermedio es mayor o menor al que buscamos. Siguiendo el anterior ejemplo:

1 2 3 5 8

, descartamos la mitad izquierda cuando consideramos que el elemento intermedio, el 3, es menor al que buscamos y solo cabrá la posibilidad de que esté a la derecha

3 5 8

5

Es sencilla su implementación por recursividad con un mínimo y tope.

Ordenación

A la hora de querer ordenar una lista o array se pueden tomar distintos algoritmos de distinta complejidad.

Burbuja - $O(n^2)$

Su nombre viene de cómo se puede visualizar el algoritmo, como una burbuja subiendo. Consiste en la permutación de valores según se encuentre que no siguen el orden indicado:

1 5 3 2 -----> 1 3 5 2

1 3 5 2 -----> 1 3 2 5

1 3 2 5 -----> 1 2 3 5

Su implementación, pese a que sencilla, resulta muy ineficiente.

Inserción - $O(n^2)$

La idea consiste en encontrar al elemento menor para colocarlo donde corresponde. Para ello se pone en el índice 0 el menor; en el 1, el menor de la posición 1 a $n - 1$; en el 2, el menor de la posición 2 a $n - 1$...

1 5 3 2

1 2 3 5

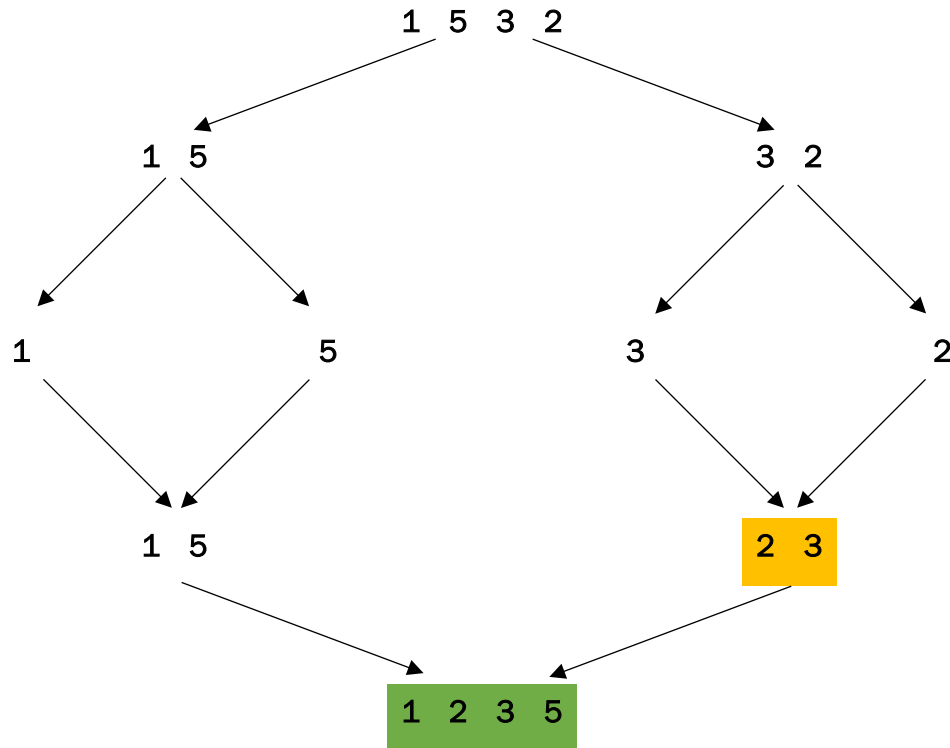
1 2 3 5

1 2 3 5

En este caso, la tercera y cuarta posición quedan ordenadas con un solo cambio. Resulta más eficiente, aunque es posible alcanzar un mejor rendimiento con las siguientes técnicas.

Mergesort - $O(n \cdot \log n)$

Basado en la psicología de divide y vencerás, consiste en realizar la llamada a la función hasta que se traten una o dos posiciones las cuales se insertarán de forma ordenadas:



Quicksort - $O(n \cdot \log n)$

También basado en divide y vencerás consiste en el uso de un pivote que se desplazará hasta la posición en la que se encuentre un valor menor. Su rendimiento es de los mejores de no estar el array casi ordenado.

Fuerza bruta

Consiste en la comprobación de todas las combinaciones posibles según sea el problema que se quiera afrontar. Son, por lo general, las soluciones más ineficientes, mas pueden ser necesarias para determinadas situaciones.

Backtracking (marcha atrás)

La marcha atrás consiste en una combinación de la recursividad con la fuerza bruta para explorar las combinaciones con cierta inteligencia. El objetivo es recorrer las posibilidades hasta determinar que es solución o no, siendo este último caso una forma de descartar todas las combinaciones que siguen a esta.

Un caso sencillo es el de la resolución de un sudoku, si vamos insertando elementos en las casillas no fijas de arriba abajo, izquierda a derecha, podremos encontrarnos con casillas que no pueden cumplir las restricciones del sudoku, en cuya ocasión volveremos a la anterior casilla no fija para probar con otro valor.

Programación Dinámica

Su nombre puede asustar de primeras, pero de alguna forma tuvieron que llamarla. Si te has dado cuenta, la función de Fibonacci es considerablemente ineficiente si se aplica recursividad, esto es porque la función repite muchas llamadas sin guardar ese valor para más tarde. La programación dinámica propone guardar estos valores en una tabla combatiendo la complejidad en tiempo con espacio en memoria.

Para el caso de Fibonacci tendríamos un ejemplo sencillo a continuación:

0	1	2	3	4	5	6	7	8	9
0	1	1	2	3	5	8	13	21	34

De forma que cuando se llame a Fibonacci de 10 solo se accederá a las posiciones del array 8 y 9 para realizar la suma y guardar este nuevo valor en la tabla.

En el caso recursivo esta llamada supondría que Fibonacci de 10 llame a Fibonacci de 8 (que a su vez llama a Fibonacci de 6 y 7, que a su vez ...) y Fibonacci de 9 (que a su vez llama a Fibonacci de 8 y Fibonacci de 7...). Creo que se entiende.

Algoritmos Voraces (Greedy)

Aunque pueden parecer tramposos, los algoritmos voraces buscan soluciones en problemas de optimización de forma rápida, pero sin seguridad de ser la óptima. Estas soluciones pueden resultar de utilidad para aplicarlas en otros algoritmos y disminuir las combinaciones que se exploran.

Divide y Vencerás

A pesar de no ser una técnica usada en la computación, en ocasiones, ver nuestro problema como pequeños problemas puede hacer aparecer una resolución más eficiente y sencilla.

Es obvio ver aplicada esta técnica cuando se divide, literalmente, el problema en dos como con las ordenaciones anteriormente mencionadas.

Librerías

Unistd

Write

```
unsigned int write(int fd, const void *buf, unsigned int n);
```

Escribe en la salida descrita por fd, n bytes desde el búfer que se le pasa.

Si queremos escribir un mensaje constante deberemos usar las dobles comillas, para generar un array temporal que pueda interpretar la función.

La función write escribe caracteres, aunque pusiéramos 4 bytes para que imprimiera un **int**, el resultado sería el de cada uno de los bytes convertidos a caracteres.

Stdio

Printf

```
int printf(const char *str, ...);
```

Imprime una cadena de caracteres y en el caso de haber más argumentos, los convierte en orden según el formato que se indique. Los formatos se expresan como un carácter que va precedidos por un '%':

Carácter	Representa
d, i	Entero en base 10
c	Carácter
s	String (char*)
f, e, ...	Double
y más ...	

La cadena se almacena en un búfer de 256 bytes que se libera cuando llega a esa cantidad de caracteres o con un salto de línea.

```
printf("Hola\n");
```

¡Atención!

De no haber salto de línea, se esperará al final de la ejecución del programa. Si este no termina debido a un bucle infinito el mensaje no aparecerá.

Stdlib

Malloc

```
void *malloc(unsigned int bytes);
```

Reserva n bytes de memoria y devuelve un puntero de tipo indefinido a la memoria reservada o **NULL** en caso de error. Su uso suele venir acompañado de la técnica de **casting** y la función **sizeof**, por ejemplo:

```
char *str = (char *) malloc (sizeof(char) * 15);
```

Free

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

Libera la memoria apuntada desde el puntero. No devuelve entero en caso de error, sino que para la ejecución del programa.

Toda memoria auxiliar que se reserve con malloc debe ser liberada para no ocasionar una **fuga de memoria**.

Atoi

Convierte una cadena de caracteres en número entero (**int**). Ejemplo:

```
int x = atoi(" +456"); // 456
int y = atoi(" -456"); // -456
int z = atoi(" + 456"); // 0
int o = atoi(" +456a213"); // 456
```

Un uso sencillo es el de pasar de argumentos como cadenas de caracteres a número.

NULL

Es una **directiva de preprocesador** que sirve para identificar un valor no válido de puntero, es equivalente a **0**. Se define en stdio y stdlib.

String

Strcpy

```
char *strcpy(char *dest, char *src);
```

Copia en la primera cadena, la segunda y devuelve el puntero de la cadena destino.

Curiosidad

Es considerablemente insegura y fue explotado por los primeros virus.

Strlen

```
unsigned int strlen(char *str);
```

Mide la longitud de la cadena, es decir, cuenta caracteres hasta encontrar el carácter nulo.

Strcat

```
char *strcat(char *dest, char *src);
```

Concatena a la primera cadena, la segunda.

Strdup

```
char *strdup(char *str);
```

Copia la cadena dada en un nuevo espacio de memoria reservado, al que apunta el puntero que se retorna por la función.

Errno

La librería de `errno` ofrece herramientas para el control de errores en las llamadas al sistema. Para ello introduce una variable global llamada `errno`, cuyo valor podrá ser actualizado tras tales momentos, codificando en esta variable un código de errores, cuyo mensaje se puede obtener con la función `stderr`.

Memoria Persistente. Archivos

La memoria de nuestro programa / proceso está asignada de forma temporal mientras este siga funcionando, pero si queremos hacer uso de esta información más tarde, deberemos guardarlo en una memoria que persista, es decir, en el disco de almacenamiento en forma de archivos.

Open

Se usa para la apertura de un archivo:

```
int open(const char *nombre_del_archivo, int flags);
```

, devuelve un file descriptor (fd), identificador del archivo, o -1 en caso de error.

Es posible dar un tercer argumento para especificar el modo en el que se crea ese archivo. Tanto las flags como modos se dan en forma de máscara de bits, por lo que, si queremos especificar varias se usará la operación OR (|).

Close

Tras usar un archivo, se debe cerrar el mismo para que se pueda hacer uso del recurso:

```
int close(int fd);
```

, devuelve 0 si se cierra correctamente o -1 en caso de error.

Read

```
unsigned int read(int fd, void *buf, unsigned int n);
```

Lee n bytes en el buffer de la entrada apuntada por el file descriptor y devuelve el número de bytes leídos o -1 de haber error.

Leer 0 nos indicará que hemos llegado al final del archivo, aunque existen otros métodos como usar EOF (End Of File).

Para leer la entrada estándar podríamos usar:

```
read(0, buf, n);
```

Write

Usado con la misma función de la librería `unistd`, pero con un file descriptor como argumento.

Compilación

La compilación en C es la conversión de los archivos de texto a binario, es una traducción que pasa por distintos procesos hasta dejar nuestro programa como instrucciones que puede interpretar el ordenador. Este proceso previo a la ejecución no es necesario en otros lenguajes, interpretados, como Python o Javascript.

Archivos de texto	A.c	B.c	D.c
Compilación (y preprocesado)	↓ (A'.c)	↓ (B'.c)	↓ (D'.c)
Archivo de objeto	A.o	B.o	D.o
Enlace	↓	↓	↓
Archivo binario ejecutable	Z.exe		

Entre los procesos de compilación se incluye la sustitución de las predirectivas de procesador.

Makefiles

Los makefiles son archivos de texto cuyo contenido expresa los archivos, flags, compilador, etc... En la práctica son scripts orientado a la compilación de programas de C.

A parte de la facilitar la compilación, nos permite evitar que se recompilen archivos que no habían cambiado (algo que puede reducir sustancialmente el tiempo de compilación en grandes proyectos) de forma sencilla.

Las variables vienen descritas por **NOMBRE=**, mientras que las reglas se escriben como **REGLA:.**

Un ejemplo de cómo puede ser un Makefile es el siguiente:

```
CC = gcc
CFLAGS = -Wall -Werror
main:
    $(CC) $(CFLAGS) main.c
```

Para ejecutarlos se usa el comando **make**, y la instrucción que se desea ejecutar. De no especificarse se escogerá la primera en el orden en el que están escritos.

Es recomendable el uso de variables, así como reglas auxiliares para la transformación de recursos y objetos.

El carácter **@** sirve para silenciar un comando en su ejecución, su utilidad es más palpable cuanto más grande es el proyecto.

Cuando añadimos parámetros al lado de la regla, indicamos qué objetos deberían estar presentes para que se ejecute dicha regla.

Reglas generales

NAME = a.out

all: \$(NAME)

\$(NAME): \$(OJBS)

Compilar o archivar los archivos objeto (.o)

clean:

Elimina los archivos objeto, .o.

fclean: clean

Elimina todos los archivos temporales y el ejecutable o librería.

re: fclean all

.PHONY: re fclean clean ...

Prioriza las reglas del makefile frente a posibles comandos o archivos del sistema.

Compilando con dependencias

Si queremos, por ejemplo, añadir la libft y que no haya relink deberemos indicar la ruta del libft.a y hacer que esa ruta sea una regla a ejecutar para compilarla.


LIBFTPATH = ./lib/libft/libft.a

(NAME): \$(OJBS) \$(LIBFTPATH)

Reglas de compilación

\$(LIBFTPATH):

make -C lib/libft # Compilar LIBFT

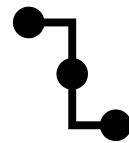


Estructuras de datos

Existe una gran variedad de estructuras de datos que nos pueden servir de utilidad, así como diversas formas de implementarlas, aunque normalmente se distinguen entre implementaciones con arrays y nodos enlazados. A continuación, se encuentra un resumen de las más usadas.

Listas

Es la estructura de datos más parecida a un array, con la diferencia de que el tamaño de esta no se ve fijado. Sus operaciones pueden ser set (cambiar valor), add (añadir nodo), remove (eliminar nodo) ...

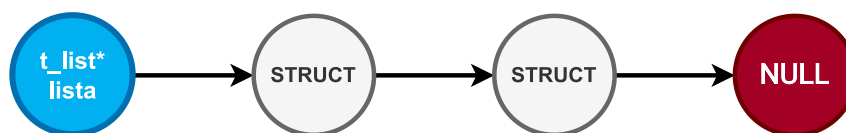


```
struct s_node{  
    void *data;  
    s_node *next;  
};  
typedef struct s_node t_list;
```

De tal modo, que si tenemos una variable:

```
t_list* lista;
```

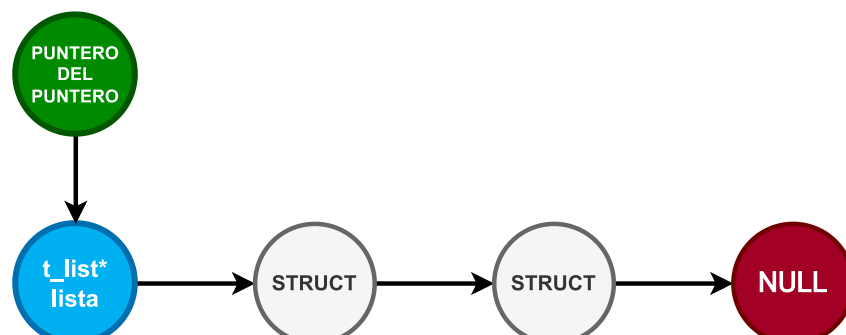
Podemos visualizar lo que tenemos como:



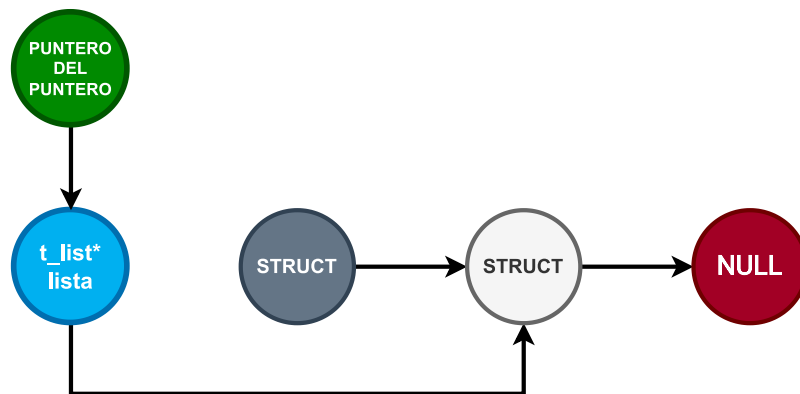
No obstante, esto podrá no ser suficiente en determinados casos, en los que tenemos que usar:

```
t_list** p_mi_lista;
```

¿Para qué?, podrás preguntarte. Si queremos que una función actualice cómo está organizada la lista, será necesario para actualizar el exterior desde la función, pasarle la dirección de memoria, como cuando se hacía con una variable int.



Un ejemplo sencillo es el de la eliminación del primer nodo de la lista.



Si le pasamos el puntero a struct a la función, la lista seguirá apuntando a esta posición que ha sido liberada y se convertirá potencialmente en un segmentation fault.

Por ello será necesario pasarle el puntero del puntero, para que se modifique hacia donde apunta.

Pilas (Stack)

Si bien lo más posible es que empiece sonándote a un conejo de las baterías, la pila, o stack en inglés, es una estructura de datos donde se van apilando los datos de forma que el primero que sería eliminado es el último que entró (**LIFO**, Last In First Out).



Sus operaciones más características son:

- **Pop**, saca el primer elemento de la pila.
- **Push**, inserta un elemento al principio de la pila.
- **Peek/Top**, devuelve, sin eliminar, el primer elemento de la pila.

Colas (Queue)

Las colas son una mezcla entre las pilas y las listas, que funcionan como una lista de espera en la vida real (**FIFO**, First In First Out), es decir, se espera (inserta) desde el final de la cola y se atiende (elimina) al primero.



Sus operaciones más características son:

- **Peek / Dequeue**, saca el primer elemento de la cola.
- **Enqueue**, inserta un elemento al final de la cola.

Para pensar

¿Cómo hacer que una cola funcione como una rueda?

Árboles (binarios)

Los árboles son estructuras de datos no lineales, en los que cada nodo puede tener uno o más subárboles. Por lo general, hablamos de árboles binarios para el uso de nodos donde distinguimos:

- **Raíz**, nodo sin ascendiente.
- **Vértice interno**, nodo con ascendiente y descendientes.
- **Hojas**, nodo sin descendientes.



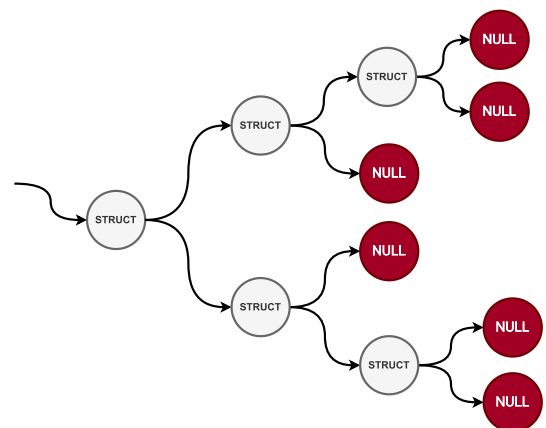
```
struct s_tree{  
    s_node *root;  
};  
  
struct s_node{  
    void *data;  
    node *t_left;  
    node *t_right;  
};
```

Cómo se insertan y modifican los nodos tras eliminaciones diferenciará los tipos de árboles, mas, en su mayoría, siguen alguna condición de orden que permite hacer más eficientes ($O(\log n)$) las operaciones con la estructura de datos.

Por ejemplo, si organizamos nuestro árbol para que funcione como un árbol BST a la izquierda los menores, a la derecha los mayores, podríamos conocer en esta estructura de datos con un menor número de accesos si un dato está o no.

Además, el recorrido que se sigue para listar los elementos de un árbol puede seguir distintas formas. Reciben el nombre de travesías y se pueden distinguir entre otras las de:

- En orden (izquierda, central, derecha)
- Pre-orden (central, izquierda, derecha)
- Post-orden (central, derecha, izquierda)
- En anchura (visitar por niveles)



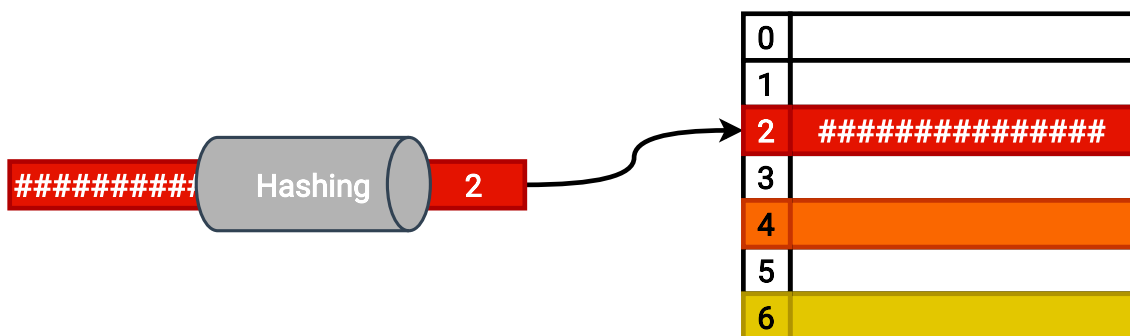
Diccionarios

Su implementación viene normalmente dada mediante árboles o tablas hash, donde cada elemento es un par de clave y valor. Las claves son únicas, a diferencia de los valores, que pueden repetirse.

Las operaciones destacadas son insert (insertar un par clave valor), get/valueOf (devolver el valor para la clave dada) ...



Conceptualmente, el funcionamiento sería el descrito con estas flechas, pero estructuralmente, se puede organizar por estructuras como tablas Hash (no ordenados) o árboles.





Misceláneo

Errores

Bus error y Segmentation Fault

Son errores procedentes de accesos ilegales a memoria. Es frecuente que se dé cuando:

- No se inicializa correctamente una variable que se pasa como argumento. Por ejemplo, se pasa **"Hola"** como argumento, pero se quiere escribir sobre ella.
- Uso excesivo (o infinito) de recursividad, frecuentemente, debido a un fallo en el caso general o base.
- No se aloja suficiente memoria en el malloc, por ejemplo, no multiplicando por el tamaño del tipo que se va a alojar. Se recomienda alojar los malloc con **sizeof**. Nótese que no es lo mismo multiplicar por sizeof(char) que de sizeof(char *).

Overflow

Aunque no se trata de un error como tal, puede conducir a una gran variedad de errores por desconocer los límites de los tipos de datos que usamos. El overflow o desbordamiento ocurre cuando realizamos una operación aritmética cuyo resultado queda fuera de los valores disponibles en el tipo que estamos tratando, dando como resultado un valor no válido.

Directivas de preprocesador

Previo al verdadero proceso de compilación se revisan las directivas de preprocesador para sustituir su contenido por el que los identificadores que se indiquen. Esto puede explotarse para funcionalidades de Debug o en proyectos de multiplataformas, donde, según el sistema operativo para el que se programe, se compile de una forma distinta, pero desde un mismo archivo.

Include

En caso de incluir una librería estándar usaremos `<*.h>`, si no, usaremos `“.h”`

Define

Parecido a una variable constante global, aunque su valor o nombre no se podrá acceder tras la compilación al haber sido sustituido por el preprocesador. Puede definirse con o sin valor:

```
#define DEBUG
```

```
#define DEBUG_LEVEL 2
```

Condicionales

Las directivas de preprocesador pueden darse con condicionales relativas a los **define**:

- **#if**, para indicar una condición que cumplir
 - **#ifdef**, para indicar que debe estar definido el identificador a su derecha.
 - **#ifndef**, lo contrario
- **#else**, indica qué hacer si no.
- **#elif**, indica qué condición comprobar si no.
- **#endif**, cierra la estructura.

Por ejemplo, los headers deben incluirse en base a un define para que el archivo solo se incluya una vez. Se sigue la siguiente convención:

```
.c
#include "mytree.h"
. . .
```

```
.h
#ifndef MYTREE_H
#define MYTREE_H
. . .
#endif
```



Cursus

Variadic function

Son aquellas que se tienen un número variado de argumentos, como es el caso de printf. Para el acceso de cada uno de ellos resulta necesario el uso de la librería stdarg, con las funciones relativas a va_list.

```
va_list // Tipo lista de argumentos
void va_start(va_list arg_ptr, prev_param);
type va_copy(va_list dest, type);
void va_copy(va_list dest, va_list src);
void va_end(va_list arg_ptr);
```

, va arg no genera memoria por lo que no debería liberarse.

Señales

Kill

Si bien buscar documentación sobre la función puede hacerle saltar las alarmas al departamento de policía más cercano, la función kill manda una señal de tipo al proceso con PID.

```
int kill(pid_t pid, int signal);
```

Recordatorio, aunque signal sea un int, lo común será hacer uso de la directiva de preprocesador que lo define.

PID

Tal vez no lo habías pensado, pero tiene bastante sentido, el PID es el ID del Proceso. Unix identifica así sus procesos, permitiendo interacciones a través del número.

signal y sigaction

La función signal es una versión reducida de sigaction. Si bien puede usarse para muchas ocasiones, si queremos especificar parámetros especiales, será necesario configurar un handler de nuestra función con sigaction. Además, podremos señalar “flags” para indicar más comportamientos que debe tener nuestro programa:

```
int signal(int signal, void (*f)(int));
```

MLX (minilibx)

Actualmente se posibilita el uso de MLX42 en el campus, por lo que el uso de la versión permitida facilita el desarrollo de este proyecto. Este repositorio viene bastante bien documentado, con ejemplos preparados para probar, incluso. Se recomienda familiarizarse con los conceptos de:

- mlx
- Textura

- Imagen
- Instancia
- Hook

Concurrencia

Una de las formas que se encontró para mejorar el rendimiento de los procesadores fue la creación de núcleos distintos que exprimían la actividad concurrente de dos procesos o hilos, si bien existían ya de antes, los avances en hardware permiten exprimirlos más. A cambio la necesidad de considerar problemas de concurrencia:

- **Exclusión mutua**, se accede a una posición de memoria que se puede actualizar. Como nuestros programas tendrán una instantánea de ese momento y otro procesador podría interrumpirnos y actualizar la variable, el valor ya no sería válido y podría ocasionar multitud de problemas. Por ello se dice que se tiene que acceder con exclusión mutua a determinadas zonas del código, ello es, no puede acceder más que uno, algo que se puede controlar con el uso de mutex o semáforos.
- **Deadlocks**, si dos mutex se pueden cancelar entre sí, el programa quedará parado ante una condición que nunca se podrá cumplir.
- **Starvation**, lo que vendría siendo, morirse de hambre, el uso de las herramientas debe ser el suficiente como para asegurar el correcto funcionamiento del programa, pero dejar que el recurso pueda ser accesible por todos. Este problema puede ser más complicado de resolver en determinadas, aunque se pueden crear mecanismos de prioridad o de aging.

En C se poseen dos mecanismos para que nuestro programa tenga concurrencia, creación de procesos (fork) y creación de hilos (pthread_create).

Crea un pequeño programa con una variable global o compartida de alguna forma, cada uno de los hilos deberá coger esta variable y aumentar su valor con postincremento (++) 1000 veces. Si revisas el valor de la variable cuando termine la ejecución de todos los hilos podrás observar que la variable no tiene el valor esperado. Esto se debe al siguiente comportamiento, donde parece que la variable se ha sumado una vez:

Pruébalo

¿Qué pasa si no usamos mecanismos para la concurrencia en nuestro programa?

-> A obtiene valor de X (x) -> X se suma en una variable de acumulador de A ($acc_a = x + 1$)

-> B obtiene el valor de X (x) -> A sobrescribe el valor de memoria ($x = acc_a$)

-> X se suma en una variable de acumulador de B ($acc_b = x + 1$)

-> B sobrescribe el valor de memoria ($x (= acc_a) = acc_b$)

Un dato que se escribe y lee por dos hilos distintos genera una **data race**.

Redirección y file descriptors

Son “claves” que permiten el acceso a ficheros abiertos del sistema. Por defecto, un programa hereda los file descriptors del padre, por lo que se suele tener 0 (stdin), 1 (stdout) y 2(stderr) abiertos.

Con dup2 podemos realizar la siguiente mecánica:

```
int fd = open("ejemplo.txt", O_RDONLY);
```

```
dup2(fd, STDOUT_FILENO);
```

N	Descripción
0	STDIN
1	STDOUT
2	STDERR
3	ejemplo.txt

N	Descripción
0	STDIN
1	ejemplo.txt
2	STDERR
3	ejemplo.txt

, y por seguridad cerrar ahora fd=3 con close() y acabar con:

N	Descripción
0	STDIN
1	ejemplo.txt
2	STDERR

, habiendo redirigido la salida estándar por el ejemplo.txt.

Pipes

Los pipes son mecanismos de comunicación entre procesos que se permiten en C. El sistema operativo crea un canal con dos

```
int canal[2];  
pipe(canal);
```

, donde canal[0] será un fd de lectura y canal[1] un fd de escritura cuyo EOF no se marcará hasta que se cierre con close (en caso de que se cree entre dos procesos).



Matemáticas

Oh, no, Matemáticas.

Números complejos

Son números compuestos por una parte real (x) y otra imaginaria (y), donde $i = \sqrt{-1}$.

$$x + iy$$

Las operaciones más básicas son:

$$\text{Suma} \rightarrow (a, b) + (c, d) = (a + c, b + d)$$

$$\text{Producto} \rightarrow (a, b) \cdot (c, d) = (ac - bd, ad + bc)$$

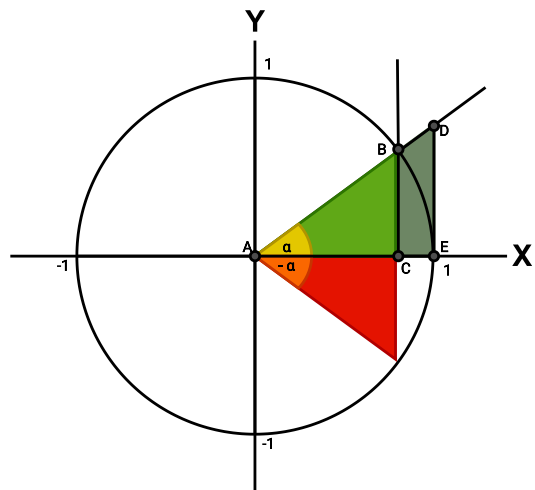
Trigonometría

Las fórmulas suelen estar bien, pero se entiende una imagen vale más. En la siguiente figura, con una circunferencia de 1 de radio y un ángulo α :

- seno (sin): B
- coseno (cos): C
- tan (tan): $\frac{B}{C} = D$

Cuando trabajamos con grados, la base es sexagesimal, algo que no se lleva muy bien con las ALUs de los operadores, por lo que se trabaja con radianes, la conversión es de:

$$\pi \text{ radianes} = 180^\circ$$



Anexo

Bits, bytes y hexadecimal

Un bit es la unidad mínima de información en la computación, si bien la agrupación de ocho de estos es la más usada, el byte u octeto. Sus valores en la computación clásica es 0 o 1 y ello se ve reflejado a nivel físico con el paso de corriente.

¿Cómo interpretar el binario?

La base binaria sigue la misma lógica que la decimal, por ejemplo:

$$25 = 2 \cdot 10^1 + 5 \cdot 10^0 = 25$$

En el caso de la base binaria la base sería de 2:

$$11001 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 25$$

¿Cómo interpretar el hexadecimal?

En la base hexadecimal se tienen 16 valores posibles 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E y F. Si bien podemos seguir el mismo patrón que con la decimal y binaria, resulta sencillo hacer la conversión entre hexadecimal y binario cuando agrupamos los bits en cuatro. Por ejemplo:

$$1001\ 1010 = 9A$$

$$0101\ 1111 = 5F$$

Escribiendo hexadecimal y octal en C

Hexadecimal

Para escribir un número en hexadecimal en C es necesario escribir 0x delante de nuestro número. Por ejemplo, la siguiente suma dará como resultado 17:

```
suma(0x10, 0x1)
```

Octal

Mientras, para escribir en octal en C tan solo es necesario escribir un 0 delante de nuestro número. Por ejemplo, la siguiente suma dará como resultado 9:

```
suma(010, 01)
```

Formatos de bases

A la hora de escribir un número, tenemos la capacidad de poner un - delante o decimales detrás sin problemas, mas a la hora de representarlo a nivel de bits es necesario definir un formato para interpretar si el número es positivo o negativo como en el tipo int o el IEEE 754 para guardar decimales.

Estos formatos tendrán sus límites de tamaño o de precisión si hablamos de decimales.

Control de fuga de memoria

Incluidas en la librería de stdlib, **atexit** y **system** son dos funciones que nos pueden ayudar a detectar fugas de memoria sin una aplicación externa. La técnica consiste en la siguiente.

Al final de nuestro programa podemos llamar a la función **atexit** pasándole como parámetro una función privada. En esta se encontrará a su vez una llamada a **system** con una cadena como parámetro "leaks nombre_del_programa".

```
void ft_leaks(void){
    system("leaks a.out")
}

int main(void) {
    . . .
    atexit(ft_leaks);
    return (0);
}
```

Notaciones

Es común y buena práctica estandarizar cómo se llaman las variables y funciones en nuestros proyectos, más aún cuando se trata de un trabajo en grupo o en una empresa, donde el código debería ser uniforme.

snake_case

Una de las notaciones más usadas consiste en el nombramiento de variables con el uso de guiones bajos. Por ejemplo:

```
char_counter
0
linked_list
```



camelCase

Otras, como camelCase consisten en el uso de mayúsculas y minúsculas para diferenciar los espacios (UpperCamelCase y loweCamelCase). Por ejemplo:



```
charCounter
0
linkedList
```

Comentarios

A pesar de que un código debería poder entenderse tal y como se lee, puede ser buena idea comentar ciertas partes del código o el principio de la función para que quede documentado.

- Comentario de una línea:

```
// Comentario
```

- Comentario definido entre una o varias líneas:

```
/* Comentario */
```

- Comentario en varias líneas:

```
/**  
 * Comentario (puede usarse @param,  
 @return o @brief) */
```

STD

STDIN - 0

Entrada estándar.

STDOUT - 1

Salida estándar.

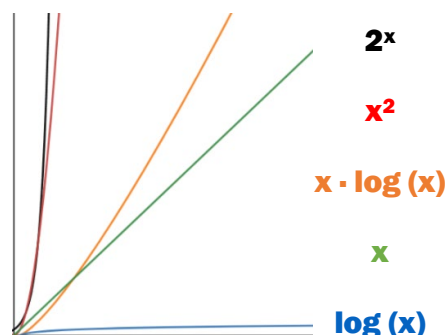
STDERR - 2

Salida de error.

La gran O - Big O Notation

Cuando queremos clasificar la complejidad temporal o espacial (E) de un algoritmo solemos usar la **notación asintótica**. Entre ellas, Θ nos indica su complejidad de forma “exacta”, mientras que la O sirve para especificar un tope (máximo) considerando los casos en los que peor rinda.

El **orden de crecimiento** indica cómo varía el coste computacional según el tamaño de la entrada:



Eficientes:

- $O(1)$ – Constante.
- $O(\log n)$ – Logarítmica.
- $O(n)$ – Lineal.
- $O(n \cdot \log n)$ – Casi lineal.

Tratables:

- $O(n^2)$ – Cuadrática.
- $O(n^3)$ – Cúbica.

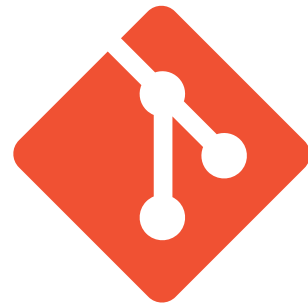
No tratables (entradas muy pequeñas):

- $O(2^n)$ – Exponencial.
- $O(n!)$ – Factorial.

Por ejemplo, las soluciones a Fibonacci que se han mostrado en este documento eran $O(2^n)$ y $O(n)$ (pero si tiene enlace y todo...). **NOTA:** Un orden de complejidad mayor no tiene por qué implicar un tiempo de ejecución mayor.

Usando Git para el control de versiones

Git es el software de control de versiones más usado actualmente. No debe confundirse con Github o Gitlab, entre otros; estos son servicios para el uso de Git distribuido que es usado para trabajar con el apoyo de la nube en grupo (o en solitario). A continuación, se resumirán algunas de las funcionalidades más cotidianas.



Commits

No existe traducción buena al español a mi parecer, aunque lo más correcto sería decir que se trata de una **confirmación**. Cuando usamos **git add** añadimos cambios que queremos que git controle y ellos serán agrupados en un commit.

Con **git status**, podemos ver el estado de estos cambios y cuáles se añadirán en el siguiente commit.

Con **git commit -m "texto"** podemos realizar el commit resumiendo los cambios en una línea de comando.

Se puede consultar el registro de commits con **git log**.

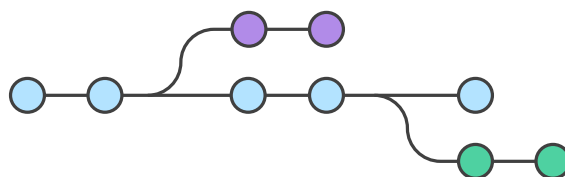
Ramas

Las ramas son líneas de desarrollo que buscan evitar el conflicto en el desarrollo de nuevas características en un proyecto.

Merge

Para añadir los cambios realizados en una rama en otra podemos usar merge. Lo primero será cambiarnos de rama con **git checkout** a donde queremos añadir los cambios y desde allí pasaremos las novedades de la rama1 con:

```
git merge rama1
```



En caso de haber conflicto, git avisará y permite al usuario qué mantener.

Ramas remotas

Pull

Consiste en la actualización del repositorio local basado en los commits remotos.

Push

Consiste en la actualización de los commits en la rama remota en base a los cambios locales.

Forks

Son clones del repositorio adecuados para cuando se quiera desarrollar una versión alternativa al proyecto que lo altere desde sus bases, pero no se quiera intervenir con el flujo de desarrollo del resto de características.

Git ignore

Es un archivo que indica a git qué archivos no debe seguir. Permite el uso de expresiones regulares y la inclusión del propio **.gitignore** como archivo a ignorar, a pesar de que no sea una buena práctica.

Por ejemplo, el contenido de nuestro gitignore podría ser:

```
.gitignore  
ex*/a.out  
ex*/main
```

Aunque puede parecer un error de gitignore, no es posible añadir un repositorio en otro, por lo que será necesario eliminar el .git. Esto será necesario según aumenta la magnitud de los proyectos.

Vi Machine (Vim)

Es un editor de texto de gran utilidad para la programación. Una motivación para seguir usándolo hoy es la carencia de interfaces gráficas comunes en terminales en servidores ahorrando así espacio y dinero para la empresa. Para escribir los comandos es necesario pulsar el botón de **ESC**, entre otros comandos tenemos:

- :x, guarda y cierra el archivo.
- :wq, escribe y cierra el archivo.
- :q!, cierra sin guardar.
- :w!, escribe en archive de solo lectura, si es posible.

Para más comandos, puedes buscar por <https://vim.rtorr.com/>.

Consejos

Doxygen Documentation

Uno de los métodos más sencillos con el cual documentar tu código siguiendo la norma (que recordará a algunos a JavaDoc).

```
/**
 * @brief . . .
 * @param var
 * @return . . . */
```

Tus archivos “cabecera” (.h), que contendrán las funciones que se usan entre unos y otros archivos deberían describir las funciones con sus parámetros y retorno. Puede que a primeras te parezca innecesario documentar cada una de tus funciones, pero te ayudará cuando tengas que volver a repasar tu código tras un descanso.

VS Code ofrece una extensión que permite empezar a escribir los comentarios si encima del prototipo de la función escribimos `/**`

Legibilidad del código

Continuamos con otra no herramienta, tus manos y tu cabeza. Recuerda que tu código es de momento tuyo, pero que para las evaluaciones o para trabajos en grupos necesitará ser entendible lo más posible. Por ello, como recomendación si la Norma no te está apretando las líneas, usa expresiones que sean fáciles de leer:

```
if (str) // str != (void *)0
    if (str != NULL)
```

```
if (!str) // str == (void *)0
    if (str == NULL)
```

Las anteriores expresiones son equivalentes en C, mas las segundas líneas son más fáciles de leer a la hora de observar el código y puede ayudarte a encontrar posibles horas que se tuvieron a la hora de considerar un caso de nulo. Además, ten en cuenta que no siempre vas a ser tú quien esté revisando tu código...

Otras expresiones como igualdades con la tabla ASCII se deberían evitar también.

```
c + ('a' - 'A')
c + 32
```

, que, si bien resulta equivalente como el ejemplo anterior, dificulta la lectura y obliga a realizar un esfuerzo de memoria.

Directivas del código

- `FUNC`, se sustituye por el nombre de la función donde se encuentra.
- `LINE`, se sustituye por el número (no cadena) del archivo donde se encuentra.
- `FILE`, se sustituye por el nombre del archivo donde se encuentra.

lldb

Ahora sí, una herramienta de verdad. El uso de `printf` o `write` puede ser rápido y útil, e incluso recomendado si quieres redirigir la salida a un archivo `.log`; sin embargo, el uso es pesado y no siempre nos da la información que estábamos buscando. Para solucionar este problema existen los **debuggers**, programas que nos permiten ver la traza de ejecución del programa con la evolución de sus variables.

lldb, es uno de ellos y se usa del siguiente modo:

1. Compilar nuestro proyecto con la flag de **gcc -g**, esto será necesario para que el debugger pueda interpretar con facilidad el archivo binario.
2. Iniciar el comando lldb con el nombre del archivo
 - a. `b <nombre / línea de la función >`, para añadir un breakpoint (punto de ruptura/parada) y nombre de la función
 - b. `run`
 - c. `gui`, (graphical user interface) interfaz gráfica, donde dar pasos con `'n'` y adentrándonos en llamadas a funciones con `'s'`.

Evitando relink

En los Makefiles que realices durante el curso deberás tener en cuenta que no se haga relink, pero ¿qué significa esto realmente? Aquí un ejemplo de qué es relink:

```
make
gcc . . .
make
gcc . . .
```

, y qué debería suceder:

```
make
gcc . . .
make
nothing to make
```

En otras palabras, si un archivo se ha compilado y no se ha variado nada en este, no debería compilarse de nuevo.

Este comportamiento suele darse por repeticiones (se menciona directa o indirectamente dos veces o más a la misma regla) o conflictos en la regla PHONY.

```
.PHONY: all fclean clean re bonus
```