Algoritmos y Programación II
Curso Buchwald/Wachenchauzer
Con lenguaje C

7 de enero de 2019

Contenidos

El lenguaje C

En este apunte se dará una introducción básica al lenguaje de programación C, asumiendo un conocimiento previo de técnicas de programación en algún otro lenguaje.

1. Características básicas del lenguaje

Se podría decir que el lenguaje de programación C es un lenguaje *sencillo*, fácil de cubrir en poco tiempo, ya que tiene pocas palabras reservadas, y una biblioteca estándar más acotada que la de otros lenguajes.

Sin embargo, la especificación actual contiene 701 páginas [1] y es posible crear código extremadamento *ofuscado* [2], de modo que no es realmente correcto decir que es sencillo.

C es un lenguaje de programación estructurado, de medio nivel, y muy portable. Esto se debe a que el modelo de computadora que usa el lenguaje se puede ajustar a una gran variedad de equipos. A veces se lo considera como un lenguaje ensamblador de alto nivel, ya que el programador suele tener que tener en cuenta detalles sobre cómo se representan los elementos del programa en la máquina, o manejar (pedir y liberar) los recursos del sistema desde el código.

2. Estándares del lenguaje

A lo largo de la historia se han desarrollado tres estándares principales. [3]

K&R El estándar publicado en la primera edición del libro "El lenguaje de programación C" de Kernighan y Ritchie.

C89 Publicado en la referencia estándar ANSI X3.159-1989 y luego en el estándar ISO/IEC 9899:1990, así como en la segunda edición del mismo libro.

C99 El estándar ISO, publicado en 1999.

Si bien a esta altura la mayoría de los compiladores de C soportan prácticamente el estándar completo de C99 [4], una gran parte de código disponible utiliza todavía el estándar C89; es por eso que en este apunte se hace especial distinción con aquellos detalles que pertencen al estándar C99.

[1] http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf

[2] http://www.ioccc.org/

[3] El último estándar del lenguaje es C11, publicado en el 2011, sin embargo todavía no tuvo un impacto notable, por lo que nos concentraremos en C99.

[4] http://gcc.gnu.org/c99status.html

3. Tipos básicos

C cuenta con una variedad de tipos numéricos. De estos, los tipos enteros pueden tomar los modificadores **`signed`** o **`unsigned`** para indicar si son o no signados.

A continuación una tabla con los distintos tipos de C, ordenados según el espacio que ocupan en memoria.

bool Se agregó en C99. Puede contener los valores 0 y 1. Incluyendo la biblioteca `<stdbool.h>`, se pueden utilizar los valores `true` y `false` (equivalentes a 1 y 0 respectivamente).

char Tipo entero, de tamaño 1 byte, la especificación no define si es **`signed`** o **`unsigned`**.

short Tipo entero, por omisión **`signed`**, debe ocupar menos espacio o el mismo que int. En el compilador gcc, arquitectura Intel 32 bits, mide 16 bits.

int Tipo entero, por omisión **`signed`**, es el tipo *natural* de la arquitectura. En el compilador gcc, arquitectura Intel 32 bits, mide 32 bits.

long Tipo entero, por omisión **`signed`**, debe ocupar igual o más espacio que long. En el compilador gcc, arquitectura Intel 32 bits, mide 32 bits.

long long Tipo entero, por omisión signed, debe ocupar igual o más espacio que long. En el compilador gcc, arquitectura Intel 32 bits, mide 64 bits.

float Tipo real, cumple con el estándar IEEE 754 de simple precisión (32 bits).

double Tipo real, cumple con el estándar IEEE 754 de doble precisión (64 bits).

long double Tipo real, según la arquitectura y las opciones de compilación, puede cumplir con el estándar IEEE 754 de doble precisión (64 bits) o de doble precisión extendida (más de 79 bits, 80 bits en arquitecturas Intel 32 bits).

complex Se agregó en C99, representa un número complejo. Ocupa dos `doubles`, y requiere incluir `<complex.h>`

float complex Se agregó en C99, de menor tamaño que el complejo común. Ocupa dos floats, también requiere `<complex.h>`.

long complex Se agregó en C99, ocupa dos **`long`** `doubles`, y también requiere `<complex.h>`.

void No se puede usar como un tipo de una variable, se usa para señalar que una función no devuelve nada o no recibe nada.

Además, se puede utilizar el modificador de **`const`** para declarar una variable que puede inicializarse pero una vez inicializada no puede modificarse.

La inicialización de las variables se realiza cuando se definen. En el caso de las funciones, los valores que reciben los parámetros actuan como inicializadores.

4. Sintáxis básica

Asumiendo conocimientos previos de programación, se describe a continuación la sintaxis básica del lenguaje de programación C.

4.1. Instrucciones

Las instrucciónes en C son lo que forman las secuencias que ejecutarán los programas, las instrucciones terminan en `;` y donde puede haber una instrucción puede haber también una serie de instrucciones entre llaves: `{` (para comenzar el bloque) y `}` (para terminar el bloque).

4.2. Valores literales

Los valores literales son valores explícitamentes escritos en el código. Y merecen un breve comentario en este resumen.

Los valores númericos se pueden escribir en decimal (`4095`), en octal (`07777`) o en hexadecimal (`0xFFF`). Además, se les puede agregar al final una `L` para indicar que es un **`long`** o una `U` para indicar que es un valor **`unsigned`**.

En el caso de los valores reales, se los puede representar con punto como separador entre parte entera y decimal o en notación científica. Por omisión, estos valores serán de tipo **`double`**, pero se puede usar una letra `F` como sufijo del valor para que se los tome como **`float`**.

Los caracteres también son valores numéricos, pero se pueden escribir a través del símbolo que representan escribiéndolos entre comillas simples. Asumiendo que se utiliza un sistema en ASCII, `'A'` será lo mismo que escribir el valor 65. Varios de los caracteres especiales (como el fin de línea) se pueden representar en C como una secuencia de `\` seguida de algún caracter, por ejemplo, el fin de línea se representa como `'\n'`.

A continuación una tabla con las secuecias que representan caracteres especiales.

Secuencia	Nombre	Descripción
`\n`	NL	fin de línea (enter)
`\t`	HT	tabulación horizontal (tab)
`\v`	VT	tabulación vertical
`\b`	BS	retroceso (backspace)
`\r`	CR	retorno de carro
`\f`	FF	avance de hoja
`\a`	BEL	señal audible (beep)
`\\`	`\`	contra barra
`\'`	`'`	comillas simples
`\0`	NUL	caracter nulo
`\ooo`	ooo	caracter con el valor octal ooo
`\xHH`	HH	caracter con el valor hexadecimal HH

Una cadena literal en C se escribe dentro de comillas dobles, por ejemplo `"ejemplo"` será un vector de 8 **`char`**, el último de estos caracteres será `'\0'` (un caracter con valor entero 0).

4.3. Estructuras Condicionales

La estructura condicional evalúa la condición, si es verdadera ejecuta el bloque verdadero, sino ejecuta el bloque alternativo.

En C, el condicional tiene dos formas básicas:

```
if (condición) {
    instrucciones;
}
```

En este caso, el bloque se ejecuta únicamente si es verdadero y si no lo es, no se ejecuta nada. La otra opción es:

```
if ( condición ) {
    instrucciones-verdadero;
} else {
    instrucciones-falso;
}
```

En ambos casos, cuando se trate de una única instrucción pueden omitirse las llaves, pero en general se recomienda utilizarlas de todas maneras para prevenir errores si luego se arreglan más instrucciones.

Una forma alternativa de la estructura condicional es la de múltiples condiciones anidadas, que suele escribirse:

```
if ( condición_1 ) {
    cuerpo_1;
} else if ( condición_2 ) {
    cuerpo_2;
} else if ( condición_3 ) {
    cuerpo_3;
} else if ( condición_4 ) {
    cuerpo_4;
} else {
    cuerpo_else;
}
```

Este tipo de estructura verifica las condiciones en cascada, hasta que una de ellas sea verdadera y en ese caso se ejecutará el cuerpo correspondiente; de no ser así, llegará al **`else`** final. Se trata únicamente de una forma de escribir cómodamente los condicionales anidados.

Otra estructura de selección multiple es el **`switch`**, que se muestra a continuación.

```
switch ( expresión_entera ) {
case valor_entero_1:
    instrucciones;
    break;
case valor_entero_2:
    instrucciones;
    break;
...
default:
    instrucciones;
    break;
}
```

En este caso, se compara la `expresion_entera` con los distintos valores enteros, y cuando coincide, se ejecutan las correspondientes instrucciones. De omitirse la instrucción **`break`**, se continúa ejecutando el siguiente bloque, sin importar que corresponda a otro valor. En el caso en que no coincida con ninguno de los valores, se ejecutará el bloque **`default`**.

Es importante notar que este tipo de selección multiple sólo puede operar con enteros, de manera que tanto la expresión usada con la instrucción **`switch`** como cada uno de los posibles valores usados con **`case`** son tomados como enteros para compararlos.

Concepto de verdadero

El concepto de verdadero de C es *todo lo que es 0 es falso, todo lo demás en verdadero*.

En C99 existe el tipo `bool` que es 0 en el caso de falso, y 1 en caso de verdadero, pero no es necesario utilizar este tipo para las condiciones, cualquier variable que valga 0 se considerará falsa, y cualquier variable con un valor distinto de 0 se considerará verdadera.

Operadores de comparación

En C existen diversos operadores de comparación entre valores, a continuación una tabla con los operadores más comunes.

Operador	Significado
`a1 == a2`	`a1` vale lo mismo que `a2`
`a1 != a2`	`a1` no vale lo mismo que `a2`
`a1 > a2`	`a1` es mayor que `a2`
`a1 < a2`	`a1` es menor que `a2`
`a1 >= a2`	`a1` es mayor o igual que `a2`
`a1 <= a2`	`a1` es menor o igual que `a2`

Además, los operadores de comparación pueden unirse o modificarse para formar expresiones más complejas.

Operador	Significado
`e1 && e2`	Debe cumplirse tanto `e1` como `e2`
`e1 || e2`	Debe cumplirse `e1`, `e2` o ambas
`! e1`	`e1` debe ser falso

Incluyendo la biblioteca `<iso646.h>` se puede usar las palabras `and`, `or`, `not`, y otros, como operadores, de la misma manera que son operadores en otros lenguajes.

4.4. Ciclos

El bucle *mientras* en C tiene la siguiente forma:

```
while ( condición ) {
    cuerpo;
}
```

La condición es evaluada en cada iteración, y mientras sea verdadera se ejecuta el cuerpo del bucle.

También existe un bucle **`do...while`**:

```
do {
    cuerpo;
} while ( condición );
```

La diferencia con el anterior es que asegura que cuerpo va a ejecutarse al menos una vez, ya que la condición se evalúa después de haber ejecutado el cuerpo.

El lenguaje C cuenta con un bucle iterativo *for*, un poco distinto a otros bucles del mismo nombre. Para comprenderlo mejor es importante notar que las dos siguientes porciones de código son equivalentes:

```
for (inicialización; condición; incremento) {
    cuerpo;
}
```

```
inicialización;
while (condición) {
    { cuerpo; }
    incremento;
}
```

4.5. Variables

Todas las variables en C hay que declararlas antes de poder usarlas, la declaración se hace de la siguiente manera:

```
tipo nombre_variable;
```

Se pueden declarar varias variables del mismo tipo separandolas con comas.

```
tipo nombre_variable_1, nombre_variable_2;
```

Además, es posible asignar un valor de inicialización al declararlas:

```
tipo nombre_variable_1 = valor_1 , nombre_variable_2 = valor 2;
```

4.6. Comentarios

En C89 la única forma de poner comentarios es utilizando bloques que comiencen con `/*` y terminen con `*/`. En C99, además, se agregó soporte de comentarios *hasta el final de la línea*, estos empiezan con `//`.

4.7. Funciones

Las funciones en C se definen de la siguiente manera:

```
tipo funcion (tipo_1 argumento_1, ..., tipo_n argumento_n)
{
    intrucciones;
    ...;
    return valor_retorno;
}
```

Es decir que el tipo que devuelve la función se coloca antes del nombre de la función, y luego se colocan los argumentos que recibe la función, precedidos por su tipo. En el caso de no recibir ningún argumento, se puede colocar simplemente `()` o `(void)`.

El cuerpo de las funciones contendrá una secuencia de declaración de variables, instrucciones, bloques, estructuras de control, etc.

Una función debe estar declarada antes (leyendo el archivo desde arriba hacia abajo) de poder llamarla en el código. Es por esto que la definición (o prototipo) de la función puede colocarse antes del contenido de la función, de forma que pueda ser utilizada por funciones que se encuentran implementadas antes. En ese caso será:

```
tipo funcion (tipo_1 argumento_1, ..., tipo_n argumento_n);
```

4.8. Punto de entrada

Se llama punto de entrada a la porción de código que se ejecuta en primer lugar cuando se llama al programa desde la línea de comandos. En C el punto de entrada es la función `main` y dado que es una función que interactúa con el sistema, tiene un prototipo en particular (con dos opciones):

```
int main (void);
```

Se puede ver que la función main devuelve un entero, que será el valor de retorno del programa, 0 indicará que el programa se ejecutó exitosamente y cualquier otro valor indicará un error. Esta opción, que no recibe parámetros, se utiliza cuando no se quieren tener en cuenta los parámetros de línea de comandos. La otra opción se utiliza cuando sí se quieren tener en cuenta estos parámetros:

```
int main (int argc, char *argv[]);
```

En este caso, los parámetros `argc` y `argv` podrían tener cualquier otro nombre, pero es convención usar estos dos. Su significado es *la cantidad de argumentos* y *un vector de punteros a los argumentos* respectivamente. Más adelante se verán en detalle los temas de vectores y punteros.

5. Tipos derivados

5.1. Vectores

Los vectores (o arreglos) son bloques continuos de memoria que contienen un número de elementos del mismo tipo. Se los declara de la siguiente manera:

```
tipo_elemento nombre_vector[tamaño];
```

Opcionalmente se puede inicializar el contenido:

```
tipo_elemento nombre_vector[] = { valor_0, valor_1, ... valor_n-1 };
```

En este caso el tamaño es implícito, el compilador lo decide a partir de la cantidad de elementos ingresada en el inicializador.

Para acceder al contenido de un vector se utiliza a través del índice del elemento dentro del vector. Los índices del vector van desde 0 hasta `largo-1`. Es importante recordar que `vector[largo]` es una posición inválida dentro del vector. Es decir:

```
tipo vector[largo];
vector[0] = valor; // asigna valor al primer elemento
valor = vector[9]; // toma el valor del décimo elemento
vector[largo-1] = valor; // asigna valor al último elemento
```

Si se accede a un vector por su nombre, sin ningún índice, se obtiene la posición en memoria del vector. Esto es una optimización para evitar tener que hacer copias de (posiblemente) grandes bloques de memoria al llamar a una función que recibe un vector. Esto tiene varias consecuencias:

- Los vectores se pasan como referencia, ya que lo que se pasa es la posición de memoria donde se encuentra el vector.

- Al recibir un vector en una función no hace falta definir el largo de este, ya que el tamaño en memoria debería haber sido definido previamente.

Esto hace que en ciertas situaciones un vector tenga un comportamiento similar al de los punteros, aunque no exactamente igual.

5.2. Punteros

Los punteros son direcciones de memoria. En C los punteros requieren tener un tipo asociado, según el tipo de datos al que apuntan (es decir, el tipo de datos que se encuentra en la porción de memoria indicada por el puntero).

El tipo `void*` se usa para apuntar a posiciones de memoria que contengan un dato de tipo desconocido.

La declaración de un puntero es igual que para una variable normal, pero se le agrega un `*` delante. Es decir:

```
tipo *puntero_a_tipo;
```

Nota: el lenguaje permite escribir el `*` pegado al tipo, también:

```
tipo* puntero_a_tipo;
```

Sin embargo las siguientes líneas son equivalentes:

```
tipo *puntero, variable;
tipo* puntero, variable;
```

En ambos casos sólo la primera variable es declarada como un puntero, la segunda es sólo una variable del tipo `tipo`.

Vale la pena aclarar que al declarar un puntero este no se inicializa con ningún valor determinado (contiene *basura*), ni se crea un espacio en memoria capaz de contener un valor de tipo `tipo`, por lo que se le debe asignar una dirección de memoria válida antes de poder operar con este.

Para obtener la dirección de memoria de un valor ya creado se utiliza el operador `&`:

```
puntero = &variable;
```

La operación contraria (*desreferenciar* un puntero) es `*`, que accede al valor referenciado por una dirección de memoria:

```
*puntero = valor;
```

Dado que en C la mayoría de las variables pasan por valor (incluyendo los punteros y con la única excepción de los vectores), si se pasa el valor de una dirección de memoria (un puntero) es posible modificar el valor referenciado por esa dirección. Por ejemplo, para leer un entero usando scanf se debe hacer:

```
scanf("%d",&entero);
```

5.3. Conversión forzada de tipos (*cast*)

La conversión forzada, o *casteo*, se utiliza para convertir un valor de un tipo a otro, cuando el compilador no es capaz de hacerlo automáticamente. Se lo logra anteponiendo un tipo entre paréntesis delante de una expresión. Por ejemplo:

```
double resultado = 3 / 2; // división entera
                          // resultado = 1.0
double resultado = (double) 3 / 2 // división flotante
                                  // resultado = 1.5
```

5.4. Estructuras

Las estructuras permiten combinar distintos tipos de datos en un mismo bloque, de la siguiente forma:

```
struct estructura {
    tipo_0 atributo_0;
    tipo_1 atributo_1;
    ...
    tipo_n atributo_n;
};
```

Esta porción de código define un nuevo tipo de datos, llamado **`struct`** `estructura`, que se puede utilizar en el resto del código.

Es importante notar que este código lleva un `;`, es uno de los pocos casos en los que debe escribirse un `;` luego de una `}`, y una fuente muy común de errores.

Las estructuras se declaran al nivel de declaraciones, (donde se definen prototipos de funciones, se incluyen encabezados, se definen enum, etc).

Una estructura ocupa en memoria por lo menos la suma de cada uno de sus atributos, además, puede haber una porción de memoria desperdiciada en la *alineación* de los datos.

Para acceder a los elementos de una estructura se utiliza el operador `.`, por ejemplo:

```
struct prueba {
    char nombre[10];
    int valor;
};
...
    struct prueba ejemplo;
    ejemplo.valor = 0;
...
```

Como todos los otros tipos de datos excepto los vectores, las estructuras en C se pasan por valor. Al trabajar con estructuras, casi siempre se utilizan punteros para pasarlas a las funciones, para evitar crear grandes copias en memoria, y para poder modificar sus atributos. Para acceder a un elemento, en ese caso, se puede escribir:

```
(*puntero_estructura).nombre
```

Como se trata de una operación muy común, esto mismo se puede escribir [5]:

```
puntero_estructura->nombre
```

[5] Esta pequeña facilidad es un poco de *azúcar sintáctico* del lenguaje

5.5. Renombrado de tipos

El operador **typedef** se utiliza para darle un nuevo nombre a un tipo existente, con la siguiente sintaxis.

```
typedef viejo_tipo nuevo_tipo;
```

Se puele utilizar **typedef** para darle un nuevo nombre a la estructura, de forma que no haga falta anteponer **struct** para usarlo, esto es:

```
typedef struct _estructura {
    tipo_1 nombre_1;
    tipo_2 nombre_2; } estructura;
```

Una vez definido de esta manera, se utiliza simplemente `estructura variable;` para declarar una variable del tipo.

En el **typedef** el nombre intermedio `_estructura` puede omitirse, pero será necesario cuando una estructura haga referencia a si misma dentro de su declaración.

5.6. Valores Enumerados

Es posible definir enumeraciones de valores enteros mediante el tipo **enum**.

```
enum dias_semana {DOMINGO, LUNES, MARTES, MIERCOLES, JUEVES, VIERNES,
                  SABADO};
enum {TRUE=1, FALSE=0, MAX_LARGO=1024};
```

En este ejemplo se define un tipo **enum** `dias_semana`, que define los valores `DOMINGO=0`, `LUNES=1` y así sucesivamente. En el segundo uso de enum no se define un tipo, simplemente se definen valores.

En el código se pueden usar los nombres de esos valores en lugar del valor en sí. Es una de las formas de *parametrizar* el código.

5.7. Uniones

Las uniones son similares a las estructuras, pero en este caso cada elemento comparte la misma ubicación en memoria.

No son muy utilizadas, pero normalmente se las usa cuando se necesita guardar un valor de distintos tipos y cada formato es excluyente (sólo uno de los tipos de datos sirve en cada caso):

```
typedef enum {ENTERO, FLOTANTE} tipo_t;

union contenedor {
    int entero;
    float flotante;
}

struct uso_prueba {
    tipo_t tipo;
    union contenedor valor;
}
```

O cuando se quiere poder tener dos forma se acceder a los mismos datos.

5.8. Asignación y Comparación

En C las asignaciones y comparaciones pueden utilizarse en cualquier parte de código, como cualquier otra expresión. Lo cual da lugar a errores, como por ejemplo un error usual:

```
while (c = 1) {
    ... // código que eventualmente modifica el valor de c
}
```

Lo que hace que un error simple se convierta en un bucle infinito en tiempo de ejecución.

Además de la asignación normal:

```
e = f // asigna el valor de f a e
```

Es C también es válido utilizar var op= valor, para obtener var = var op valor, ejemplos:

```
e += f // e = e + f
e -= f // e = e - f
e *= f // e = e * f
```

Además, cuando se debe incrementar o decrementar un valor en 1, C provee pre/post (in/-de)crementos, por ejemplo:

```
a = 0; b = 0; c = 0; d = 0;
e = a++; // a = a + 1, Post incremento, e = 0, a = 1
e = ++b; // b = b + 1, Pre incremento, e = 1, b = 1
e = c--; // c = c - 1, Post decremento, e = 0, c = -1
e = --d; // d = d - 1, Pre  decremento, e = -1, d = -1
```

Las expresiones en C propagan valores de izquierda a derecha, el valor que se propaga es el que puede ser revisado eventualmente por las estructuras **while**, **if**, **for**, etc.
Ejemplo:

```
a = b = c = d = e = f = 1; // usa la propagación para asignar varias
                           // variables a la vez.
```

5.9. Constantes

C tiene tres tipos de constantes distintos: las que se definen con el preprocesador, los tipos enumeraros y las variables con el modificador const.

Las constantes del preprocesador de C son *macros* que son reemplazados por el preprocesador, que corre antes que el compilador. El preprocesador no conoce el lenguaje, sólo busca ocurrencias de una secuencia de caracteres y las reemplaza por otras, lo cual puede ser problemático en algunas situaciones particulares. Ejemplos:

```
#define MAX_LARGO 2048
#define AUTHOR "Mi Nombre"
#define DATE "2009-09-01"
#define LICENSE "CC-3.0-BY-SA"
...
    int vector[MAX_LARGO];
```

Los valores enumerados que ya fueron mencionados anteriormente sólo pueden contener valores enteros (**int**), es la forma recomendada de tener constantes enteras, ya que es fácilmente parametrizable y no tiene las desventajas del preprocesador.

Las variables con el modificador const pueden usarse y una vez inicializadas no puede alterarse el contenido sin hacer un casteo.

6. Ejemplo básico

Desde hace muchos años este es el ejemplo básico de programación en C.

```
1 #include <stdio.h>
2
3 int main(int argc, char* argv[])
4 {
5     printf("Hola mundo\n");
6     return 0;
7 }
```

En la primera línea de este ejemplo hay una instrucción **#include**, se trata de una instrucción al preprocesador[6]. Esta instrucción significa que todo lo que está en el archivo especificado se incluye dentro del archivo actual. Los <> alrededor del nombre del archivo significan que el preprocesador debe buscar el archivo en la ruta de inclusión del sistema. Si se utilizara "" en lugar de <>, se buscará el archivo en la ruta actual de compilación.

En la práctica las <> se utilizan para incluir encabezados (conjuntos de prototipos, definiciones de tipos y constantes, etc) de las bibliotecas externas al programa que se vayan a utilizar, que se deben encontrar instaladas en el sistema. Mientras que las comillas dobles se utilizan para incluir encabezados propios de otras porciones del mismo programa.

En particular la biblioteca `stdio.h` es la biblioteca estándar de entrada y salida, en este caso es incluida para poder usar `printf` que es una función de la biblioteca estándar de C para imprimir por salida estándar (normalmente, la consola). En este caso, `printf` recibe un único parámetro que será la salida a imprimir; pero puede también recibir más parámetros, para lograr una salida más avanzada.

El primer argumento de `printf` es siempre una cadena, que puede tener un formato especial o no, indicando qué tipos de variables se deben imprimir y de qué forma. Además, puede tener marcas especiales para indicar el fin de línea (`'\n'`), una tabulación (`'\t'`), una contrabarra (`'\\'`) y algunos más.

Vale la pena notar que `printf` no es parte del lenguaje sino de la biblioteca estándar, que está especificada en el mismo estándar donde está especificado el lenguaje pero aún así, no es parte del lenguaje.

Se puede encontrar documentación completa de `printf` y de las otras funciones de biblioteca mediante las páginas del manual, generalmente instaladas en los sistemas Linux o similares (`$ man 3 printf`) o mediante el programa gráfico `yelp` en estos mismos sistemas, o bien on-line en cualquier sitio que publique las páginas de manual en internet [7] [8].

[6] El preprocesador es una herramienta que corre al compilar el programa, antes de correr el compilador, las instrucciones de preprocesador siempre comienzan con **#**

[7] http://linux.die.net/man/

[8] http://www.linuxinfor.com/spanish/man3/index.html

7. Compilación

Para poder compilar programas en C, es necesario contar con un entorno de programación que permita compilar, enlazar y correr los programas compilados. Esto requiere tener el compilador de C instalado, junto con la versión para desarrollar de la biblioteca estándar.

Existen numerosos programas [9] que permiten compilar, enlazar y correr apretando una tecla o eligiendo una opción desde un menu. Si bien estos programas son una ayuda para el desarrollador, no son indispensables, es posible editar el código del programa con cualquier archivo de texto y luego compilarlo desde la línea de comandos.

El compilador más difundido en los sistemas Linux y uno de los más difundidos en general es el compilador **gcc**. Se trata de un compilador libre, con muchos años de madurez, y es el que se explica en este apunte.

Asumiendo que el ejemplo presentado antes se grabó como `hola.c`, para compilarlo usando `gcc` será necesario escribir, en el directorio donde se encuentra el código del programa:

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

Esto generará el archivo ejecutable `hola` en ese mismo directorio. Si bien no se puede ver en esta sencilla línea de comandos, hay varios pasos involucrados en la compilación de up programa.

- En primer lugar, el código es procesado por un *prepocesador*, que se encarga de hacer los **`#include`** antes mencionados, entre muchas otras cosas.
- La salida del preprocesador es *compilada*, es decir que el código C es convertido en código binario que pueda ser ejecutado por la computadora.
- Una vez compilado, el programa es *enlazado* con las bibliotecas que utilizadas, en el ejemplo anterior con la biblioteca estándar de C, para poder usar `printf`.

Con **gcc** es posible realizar estos pasos intermedios uno por uno:

```
$ # Prepocesador
$ gcc -E hola.c -o hola.i
$ # Compilador
$ gcc -c hola.i -o hola.o
$ # Enlazador
$ gcc hola.o -o hola
$ # Ejecución del programa
$ ./hola
```

El compilador gcc tiene una gran variedad de otras opciones que se pueden consultar en las páginas de manual del mismo (`man gcc`). A continuación algunas de las más importantes.

[9] Codeblocks, Geany, Anjuta, Kdevelop, etc

Opción	Acción
`-Wall`	Muestra advertencias por cada detalle que el compilador detecta como posible error de programación.
`--pedantic`	El compilador se pone en modo pedante, busca más posibles errores de programación e interrumpe la compilación por estos.
`--std=c99`	El compilador compila código usando el estándar C99 (C89 se utiliza por omisión)
`-g`	Pone marcas en el archivo generado para que las use el *debugger* (`gdb`).
`-O` ó `-O1`	Habilita las optimizaciones básicas. Las optimizaciones pueden cambiar el flujo del programa por lo que es muy poco recomendable aplicar optimizaciones sobre un código a utilizar con un debugger.
`-O2`	Habilita todas las optimizaciones basicas y varias avanzadas que se consideran seguras.
`-O3`	Habilita todas las optimizaciones basicas y varias avanzadas, incluso las que no se consideran del todo seguras (pueden generar errores en situaciones de borde).
`-Os`	Habilita las optimizaciones que reducen el tamaño del código.
`-O0`	Deshabilita todas las optimizaciones, este es el comportamiento por omisión.

Manejo de memoria en C

Todas las variables, en el lenguaje C, se definen dentro de alguna función, fuera de esa función no es posible acceder a ellas. Al entrar a una función, a cada una de las variables definidas en esa función se le asigna el espacio que sea necesario dentro de una pila interna de memoria (*stack*) con la que cuenta el programa, y al terminar la función se desapila todo lo definido en ella. Es decir que la pila crece con cada llamado a una función y decrece con cada función que se termina.

Por otro lado, todos los tipos que C define como parte del lenguaje son de un tamaño fijo, incluso los definidos por el usuario usando **`struct`**. Es por eso que el espacio que se reserva en la pila interna de memoria tiene un tamaño fijo.

Existe, además, otro espacio de memoria que se utiliza cuando el tamaño de los datos no es fijo. A este espacio de memoria dinámica se lo llama *heap*, contiene bloques de memoria, que el programador puede solicitar para utilizar según sea conveniente.

8. Obtener el tamaño de un tipo (*sizeof()*)

El operador **`sizeof`**`()` devuelve el tamaño en bytes de un tipo de datos (como **`int`**). Por comodidad se le puede pasar tanto el nombre de una variable o el nombre de un tipo de datos, en ambos casos devolverá el tamaño del tipo de datos asociado.

```
int largo, a;
largo = sizeof(int);  // Cantidad en bytes de int
largo = sizeof(a);    // Identico a lo anterior
```

En este caso, ambas llamadas a **`sizeof`** devuelven el tamaño que ocupa un entero en memoria en la arquitectura y compilador que se esté utilizando (por lo general son 4 bytes).

```
char c;
largo = sizeof(c); // Cantidad de bytes de un char
```

En este caso, se devuelve cuánto ocupa un caracter. Los caracteres ocupan siempre 1 byte.

```
char *puntero;
largo = sizeof(puntero); // Cantidad de bytes usados por un puntero.
```

En este caso, se devuelve la cantidad de bytes que ocupa un puntero. Todos los punteros tienen el mismo tamaño, y es tal que pueda contener cualquier dirección de memoria, sea estática o dinámica.

```
int vector[100];
largo = sizeof(vector);                // Cantidad de bytes del vector
largo = sizeof(vector) / sizeof(int);  // Largo del vector (100)
```

En este caso, la primera llamada devuelve el tamaño total en bytes ocupado por el vector (usualmente serían 400 bytes), mientras que la segunda devuelve siempre 100 sin importar la plataforma, ya que divide el espacio total del vector por el tamaño de cada uno de los elementos.

> **Atención**
> Los vectores no siempre se comportan como punteros. En el ejemplo anterior `vector` no se refiere a la posición de memoria donde comienza el vector, sino a todo el vector. Este es el comportamiento esperado para los vectores estáticos definidos en el mismo entorno, esto quiere decir, que si el vector lo recibieramos en una función y le hacemos **`sizeof`**`(vector)` el resultado sería la cantidad de bytes usados por un puntero.

9. Memoria dinámica en C

Mediante la utilización de los punteros, es posible acceder a cualquier porción de memoria válida, tanto si se encuentra dentro de la pila interna como si se encuentra dentro del espacio de memoria dinámica.

Para obtener una porción de memoria válida dentro del espacio de memoria dinámica, existen en la biblioteca estándar funciones (`malloc` y `realloc`) que reservan un bloque de memoria y devuelven su dirección. Utilizando estas funciones es posible conseguir estructuras de datos dinámicas, que pueden variar su tamaño según sea necesario, en lugar de tener un tamaño ya definido.

Es importante notar que la memoria dinámica reservada mediante las funciones de la biblioteca estándar no es liberada automáticamente, quien haya hecho la reserva de memoria debe encargarse también de liberarla (con la correspondiente función de biblioteca estándar, `free`), de no ser así, se dice que el programa *pierde memoria*, ya que los bloques reservados no pueden volverse a utilizar aún cuando ya no estén en uso.

9.1. Pedir memoria al sistema (*malloc()*)

Para pedir memoria al sistema se utiliza la función `malloc`[10], definida en `<stdlib.h>`, cuyo prototipo es el siguiente:

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

Si el sistema tiene suficiente memoria disponible, `malloc` devuelve un puntero a la primera posición de memoria de un bloque de memoria dinámica, de `tamanio` bytes.

Si ocurriera algún problema, porque el sistema no dispone de suficiente memoria o similar, la llamada de `malloc` devolvería `NULL`.

El tipo de datos `size_t` es un valor entero sin signo capaz que contener cualquier tamaño de memoria válido en la arquitectura actual.

10. Devolver memoria al sistema (*free()*)

Cuando un bloque de memoria ya no es necesario para un programa, se lo debe devolver al sistema, de forma que el sistema pueda tenerlo nuevamente entre los recursos a utilizar por

[10] Para más información: `man 3 malloc`.

otros procesos. La función `free`[11] de la biblioteca estándar hace exactamente esto, su prototipo es:

```
void free(void *puntero);
```

Recibe como único parámetro un puntero antes devuelto por `malloc` o `realloc`, libera ese bloque de memoria y no devuelve nada.

Es importante notar que no es posible liberar una porción de memoria que ya ha sido liberada. Esta acción puede provocar el mismo tipo de errores que los provocados por acceder a una porción de memoria inválida.

11. Agrandar/achicar un bloque de memoria (*realloc()*)

Cuando se necesita modificar el tamaño de un bloque memoria, se utiliza la función `realloc`[12] de la biblioteca estándar. Su prototipo es el siguiente:

```
void *realloc(void *puntero_anterior, size_t nuevo_tamanio);
```

Recibe un puntero antes obtenido mediante `malloc` o `realloc` y el nuevo tamaño del bloque de memoria. Si todo funciona bien, devuelve un nuevo puntero al nuevo bloque de memoria, copia el contenido del bloque viejo al nuevo (copia el largo mínimo entre los dos bloques), y el bloque anterior es liberado.

Si algo falla, devuelve `NULL`, y el bloque anterior no se modifica.

Teniendo en cuenta este comportamiento, normalmente **no** se utiliza una construcción como la siguiente.

```
/* MAL */
datos = realloc(datos, sizeof(dato_t) * tamanio_nuevo);
```

Ya que si `realloc` no puede cumplir lo pedido devolverá `NULL` y entonces existirá en memoria un bloque de datos válido, con información válida, al cual es imposible acceder, ya que se perdió la dirección de memoria que antes estaba en `datos`.

En cambio se debe utilizar:

```
void *aux = realloc(datos, sizeof(dato_t) * tamanio_nuevo);
if ( aux == NULL ) {
    // realloc no pudo pedir el bloque nuevo, hacer algo al respecto.
} else {
    datos = aux;
}
```

Otro posible problema a tener en cuenta es cuando se tienen punteros que apuntan a **partes** de un bloque de memoria y se llama a `realloc` sobre ese bloque: los punteros pasarán a ser inválidos, ya que apuntan a direcciones que ya no son las del bloque en cuestión.

De modo que hay que tener mucho cuidado de nunca guardar referencias a porciones de memoria que pueden ser movidas de lugar mediante `realloc`.

[11] Para más información: `man 3 free`.

[12] Para más información: `man 3 realloc`.

12. Ejemplo: una pila de tamaño variable

Tener una pila de tamaño variable es un ejemplo sencillo de manejo de memoria dinámica. La pila contiene un arreglo de valores, donde se van apilando y desapilando los elementos; el problema se da cuando se quieren apilar más elementos que los que la pila puede almacenar, en ese caso se debe reservar un bloque de memoria de mayor tamaño para que siga siendo posible agregar elementos a la pila.

En este caso, la estructura de la pila será de la forma:

```
struct _pila {
    void **datos;
    size_t tam;
    size_t largo;
};
typedef struct _pila pila_t;
```

Donde `cantidad` es la cantidad de elementos almacenada en la pila, mientras que `tamanio` es el tamaño actual de la pila, es decir, la máxima cantidad de elementos que puede almacenar antes de tener que reservar una porción mayor de memoria.

12.1. Creación de la pila

Cuando se trabaja con memoria dinámica, las funciones de creación de una estructura, no sólo deben inicializar los atributos de la estructura, sino que también deben hacer el primer pedido de memoria, para reservar el bloque inicial con el que se trabajará.

```
pila_t *pila_crear()
{
    // Pedido de memoria para la pila
    pila_t *pila = malloc(sizeof(pila_t));
    if (!pila) return NULL;

    // Pedido de memoria para los datos de la pila
    pila->datos = malloc(MIN_TAM * sizeof(void *));
    if (! pila->datos) {
        free(pila);
        return NULL;
    }

    // Otras inicializaciones
    pila->largo = 0;
    pila->tam = MIN_TAM;
    return pila;
}
```

La función `malloc` es la encargada de reservar un bloque de memoria para la estructura de la pila y luego para el bloque de datos que contendrá la pila. El tamaño de la estructura pila lo obtenemos con **`sizeof`**`(pila_t)`, mientras que el tamaño para el bloque inicial de datos es el resultado de multiplicar una constante `MIN_TAM` por el tamaño del tipo de dato que contiene la pila (**`void`** `*`, es una pila de punteros), es decir que en primera instancia la pila podrá hasta contener `MIN_TAM` elementos.

Si por algún motivo el sistema operativo no pudiera reservar la memoria requerida, la función `malloc` devuelve `NULL`. En ese caso, la función de creación de la pila devuelve `NULL` para indicar que no se ha podido crear la pila, si falla el segundo malloc, debemos liberar el bloque de memoria pedido por el primero malloc.

12.2. Incremento del tamaño de la pila

En el caso de agotarse el lugar, será necesario reservar una porción mayor de memoria. Es decir que la función `apilar` deberá ser de la forma:

```
bool pila_apilar(pila_t *pila, void* valor)
{
    if (!pila) return false;

    // Verifica si hay que agrandar, si no puede devuelve false.
    if (pila->largo == pila->tam) {
        if (! pila_redimensionar(pila, 2 * pila->tam)) {
            return false;
        }
    }

    // Asigna el valor y avanza
    pila->datos[pila->largo] = valor;
    pila->largo++;
    return true;
}
```

En esta función, cuando la cantidad de elementos es igual o mayor al tamaño actual de la pila, se llama a la función `pila_redimensionar`, que será la encargada de reservar un bloque de mayor tamaño, en este caso se le pide que el bloque sea del doble del tamaño original. La función `pila_redimensionar` tendrá la siguiente forma:

```
bool pila_redimensionar(pila_t *pila, size_t nuevo_tam)
{
    // No achica la pila menos del valor inicial.
    if (nuevo_tam < MIN_TAM) nuevo_tam = MIN_TAM;

    // Pide la nueva memoria
    void **nuevo = realloc(pila->datos, nuevo_tam * sizeof(void *));
    if (! nuevo) return false;

    // Asigna los nuevos valores
    pila->datos = nuevo;
    pila->tam = nuevo_tam;
    return true;
}
```

En este caso se utiliza la función `realloc`, que recibe la dirección actual donde se encuentran los datos, y el nuevo tamaño que se quiere reservar. `realloc` se encarga de reservar el nuevo bloque, copiar toda la información que estaba en el bloque viejo al nuevo y liberar el viejo.

Al igual que en el caso de la creación de la pila, si por algún motivo no es posible reservar la memoria según se quiere, `realloc` devuelve `NULL`. En este caso, los valores que ya estaban en la pila siguen estando ahí, simplemente significa que no se ha podido agrandar la porción de memoria reservada según se había pedido, es por ello que se utiliza un puntero auxiliar `nuevo` y sólo se lo asigna al atributo `datos` en el caso en que la reserva de memoria haya sido exitosa.

12.3. Destrucción de la pila

Como ya se dijo, cuando se reserva memoria mediante estas funciones, es importante luego liberar la memoria reservada, porque de no hacerlo quedan bloques de memoria inutilizables. Es por ello que será necesario contar con una función `pila_destruir`, y quien utilice la pila deberá recordar llamar a esta función al terminar de utilizarla.

```
void pila_destruir(pila_t *pila)
{
    if (pila) free(pila->datos);
    free(pila);
}
```

Una vez que se ha llamado a la función `free`, la porción de memoria dinámica deja de estar reservada, ya no es más una porción de memoria válida y no puede ser accedida por el programa (a menos que se haga una nueva reserva).

12.4. Disminución del tamaño de la pila

Finalmente, si bien es posible tener una pila que sólo crezca y nunca se reduzca, en general es deseable liberar la memoria que no está siendo utilizada, para que pueda ser usada por otras partes del programa. De modo que sería deseable que la pila se reduzca al desapilar, cuando el espacio ocupado por los elementos en mucho menor que el tamaño de la pila.

```
void* pila_desapilar(pila_t *pila)
{
    if (pila_esta_vacia(pila)) return NULL;

    // Desapila y se guarda el tope.
    pila->largo--;
    void *r = pila->datos[pila->largo];

    // Verifica si hay que achicar, si no puede no hace nada
    if ((pila->tam > MIN_TAM) && (pila->largo <= (pila->tam/4))) {
        pila_redimensionar(pila, pila->tam / 2);
    }

    return r;
}
```

En este caso, luego de desapilar el elemento pedido, la función verifica si la cantidad de elementos ocupa menos de un cuarto del tamaño total de la pila, y de ser así, reduce el tamaño a la mitad.

13. Aritmética de punteros

Las direcciones de memoria en C son valores enteros positivos, el valor `NULL` es equivalente a `0`, que es una posición de memoria inválida.

Los punteros contienen direcciones de memoria asociadas a un tipo en particular (a excepción de el tipo **`void`**). Como ya se vio, cada tipo tiene asociado un tamaño en bytes. El compilador de C utiliza esta información para poder realizar operaciones aritméticas (sumas y restas de valores enteros) con punteros.

Este es un tema que se presenta en principio complejo, pero que hace que se pueda operar de forma muy poderosa sobre las porciones de memoria utilizadas. En particular, es posible utilizar esta técnica para recorrer un arreglo de valores, sin que necesariamente se los haya declarado como vector. Ejemplo:

```
float fs[MAX_LARGO];
float *pf = fs;            // pf apunta al comienzo del vector fs
pf = pf + 1;               // pf apunta al segundo elemento de fs (&fs[1])
pf = fs + MAX_LARGO - 1;   // pf apunta a el último elemento de fs
```

En este caso se declara un vector de valores de tipo **`float`** (que ocupan 4 bytes cada uno), luego se declara un puntero a valores de tipo **`float`**, que se inicializa con la posición de memoria del primer elemento del vector.

Al sumarle 1, sin embargo, la posición de memoria se incrementa 4 bytes, ya que se trata de un puntero y de esta manera avanza al siguiente **`float`** del vector.

De la misma manera, en la última línea, se obtiene un puntero a la dirección de memoria del último elemento del arreglo.

Si no se declara el puntero del tipo correcto, en cambio, no es posible operar de esta manera con las direcciones de memoria. Es decir que si se hiciera algo como lo siguiente:

```
void *pv = fs;  // pv apunta al comienzo del vector fs (sin tipo asociado)
pv = pv+1;      // pv apunta al segundo byte de fs *ERROR*
```

Se obtendría un puntero a la dirección de memoria del segundo **byte** del vector de **`float`**, lo cual podría dar lugar a diversos errores, ya que si se quiere acceder a la información, se estaría accediendo a 3 bytes de un valor y 1 byte del siguiente.

Es por ello que es posible decir que el operador de acceso a un elemento `[]` en C es *azúcar sintáctico*[13], siendo `a[10]` sintácticamente equivalente a `*(a+10)`, así como a `10[a]`. Claro que este último, si bien válido, hace que el código sea extremadamente poco legible, por lo que no se lo debe utilizar.

14. Uso directo de bloques de memoria

En la biblioteca estándar de C hay varias funciones útiles que acceden directamente a la memoria, que permiten copiar o inicializar valores, que, por lo general, serán necesarias cuando se trabaje con bloques de memoria.

Estas funciones asumen que tanto el bloque de memoria origen y destino son porciones válidas de memoria, y que pueden ser escritas desde el programa. De no ser así, el sistema

[13] Un agregado a la sintaxis del lenguaje para hacerla más agradable, pero no imprescindible.

probablemente termine la ejecución del programa al encontrar un acceso a una porción de memoria inválida, generalmente mediante el error *Violación de segmento* (*Segmentation fault*).

Además, es importante tener en cuenta que todas estas operaciones tienen un costo lineal con respecto al tamaño de memoria sobre el cual operan, es decir que el tiempo requerido para ejecutarlas depende del tamaño de los bloques de memoria.

14.1. Copiar contenidos de bloques de memoria (*memcpy()* y *memmove()*)

Para copiar el contenido de un bloque memoria a otro se puede utilizar la función `memcpy`[14], declarada en el encabezado `<string.h>`, cuyo prototipo es:

```
void *memcpy(void *destino, const void *origen, size_t cantidad);
```

La función copia `cantidad` bytes desde la posición de memoria `origen` hacia la posición `destino` y devuelve un puntero a `destino`. La función asume que origen y destino son bloques que no se solapan.

Cuando origen y destino se solapan se debe utilizar la función `memmove`[15], también declarada en el encabezado `<string.h>`, cuyo prototipo es:

```
void *memmove(void *destino, const void *origen, size_t cantidad);
```

La función copia `cantidad` bytes desde la posición de memoria `origen` hacia la posición `destino` y devuelve un puntero a `destino`. De haber solapamiento, se encarga de que no se pierdan los datos al momento de hacer la copia.

14.2. Inicialización de un bloque de memoria (*memset()*)

Para inicializar un bloque de memoria con un valor se puede utilizar la función `memset`[16], definida en el encabezado `<string.h>`, cuyo prototipo es:

```
void *memset(void *direccion, int byte, size_t cantidad);
```

Que escribe el valor de `byte`, en el bloque de `cantidad` bytes, que empieza en `direccion`. Devuelve un puntero a `direccion`.

[14] Para más información: `man 3 memcpy`.
[15] Para más información: `man 3 memmove`.
[16] Para más información: `man 3 memset`.

Vectores y punteros

Una particularidad de C, que puede parecer poco intuitiva al principiante, es que al utilizar el nombre de una variable de un vector en nuestro código, C (casi siempre)[17] usa la dirección de memoria donde está ubicado el vector. Es decir, para prácticamente todos los usos, el nombre de un vector es equivalente a un puntero.

El siguiente código imprime la posición de memoria del vector a:

```
int a[] = {1, 2, 3, 4, 5};
printf("%p\n", a);
```

Por otro lado,

```
int *p = a;
```

Es una intrucción válida, ya que C guarda la dirección de memoria de a en p. Sin embargo, esto no quiere decir que un vector sea un puntero.

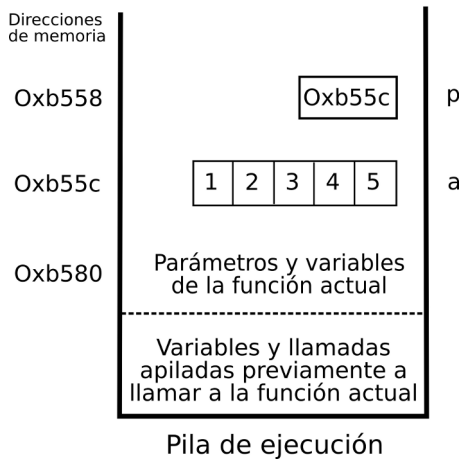

Figura 11.1: Pila de ejecución del código mostrado

Además de almacenarse de forma distinta en memoria, a un vector no se le puede cambiar su posición en memoria, por lo que el siguiente código es inválido:

```
int b[10];
b = a; // ESTO NO ANDA
```

En una función que recibe un arreglo por parámetro, en realidad C le pasa la dirección de memoria del vector. Por ejemplo, si definimos la siguiente función `suma`:

[17] Excepto para sizeof, & y en el uso en la inicialización de un vector en la declaración.

```
long suma(int *datos, int largo)
{
  long res = 0;
  for(int i = 0; i<largo; i++) {
      res += datos[i];
  }
  return res;
}
```

El puntero `datos` tendrá la dirección de memoria del vector recibido, `datos` es un puntero y no un vector. Para invocar a la función, podremos hacerlo de la siguiente forma:

```
suma(a, 5);
```

Siendo `a` el mismo vector con el que venimos trabajando. Al poner el nombre del arreglo, C usa la dirección de memoria de `a` en la invocación a la función `suma`.

En el código de `suma` vemos que el puntero se usa exactamente igual que como usaríamos el vector. Nuevamente, C usa la dirección del arreglo (incluso para el operador `[i]`), por lo que el uso es casi idéntico a utilizar un puntero.

En particular, el operador `vector[posicion]` es exactamente lo mismo que escribir `*(vector+posicion)` y esto funciona ya que al sumar un entero a una dirección de memoria el entero se multiplica por el **`sizeof`** del tipo apuntado (a esto último se lo suele llamar *aritmética de punteros*).

Para mayor claridad, el tipo de variable recibida (**`int`** `*datos`) se puede escribir también como **`int`** `datos[]`, que es la forma usual de recibir un vector en C.

15. Vectores de vectores

Al declarar cualquier tipo de vector, C requiere que cada posición del vector tenga exactamente el mismo largo (en bytes), por lo que cada posición debe ser de un tamaño fijo.

Es por ello que, si se desea armar un vector de vectores (una matriz), se lo deba hacer de una forma particular. Por ejemplo, si se desea construir una matriz de 3 filas y 4 columnas, se lo podría hacer de la siguiente manera:

```
int v[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
```

Como se ve, es posible no especificar la cantidad de filas, ya que estas se especificarán automáticamente al inicializar, pero es imprescindible especificar la cantidad de columnas, ya que esto es lo que determina la medida de cada uno de los elementos del vector `v`.

Para acceder a la información del vector, se lo hará de la forma intuitiva: `v[fila][columna]`.

Hasta aquí no resulta demasiado complejo. Las limitaciones surgen cuando se quiere poder pasar un vector por parámetro a una función. La forma correcta de pasar un vector como el mostrado a una función, sería la siguiente:

```
void imprimir_matriz(int v[][4], size_t filas);
```

Es decir que la cantidad de filas puede ser variable, y se la debe recibir por parámetro, pero la cantidad de columnas es parte del tipo de la variable, y se la debe indicar dentro de la declaración de la función.

Esto se debe a que los elementos en memoria se guardan uno a continuación del otro, y C necesita saber cuántos elementos tiene cada una de las filas para poder saber cuánto se tiene que desplazar hasta encontrar el elemento.

16. Usar un bloque contiguo de memoria como una matriz

Una forma de poder hacer un manejo genérico de matrices en C es basarse en que la matriz está guardada en forma contigua en memoria y hacer la cuenta de la posición que le corresponde a cada elemento.

Siguiendo con el ejemplo anterior, `v` como dirección de memoria es la dirección del elemento `v[0][0]`. En el siguiente espacio dentro del bloque de la memoria, que se encuentra desplazado la cantidad de bytes que mide un entero estará `v[0][1]`. Desplazándose cuatro veces ese tamaño tenemos `v[1][0]`.

Utilizando esta idea, podemos crear una nueva variable `x`, que sea un puntero a enteros que apunta al comienzo de `v`. Podemos utilizar la *aritmética de punteros* vista anteriormente: haciendo `(x + fila*cols)` (donde `cols` es la cantidad de elementos por fila) obtenemos la dirección del comienzo la fila `fila`, esta dirección es el comienzo de un vector de enteros, por lo que podemos hacer `(x + fila*cols)[columna]`, para acceder al valor de `v[fila][columna]`.

La ventaja que obtenemos al hacer esto es que podemos escribir funciones genéricas, que reciban el tamaño de la matriz por parámetro. Por ejemplo:

```
1 void imprimir_matriz(void* aux, size_t filas, size_t cols)
2 {
3     int *x = aux;
4     for (int i=0; i < filas; i++) {
5         for (int j=0; j < cols; j++) {
6             printf("%d ", (x + i*cols)[j]);
7         }
8         printf("\n");
9     }
10 }
```

Y la invocación a esta función sería `imprimir_matriz(v, 3, 4)`

17. Vectores de vectores de tamaño variable

Teniendo en cuenta lo visto anteriormente, si se quiere crear matrices cuyas columnas sean variables como las filas, que se las pueda pasar por parámetro a funciones sin importar cuántas columnas tenga, será necesario recurrir a la memoria dinámica.

Para el caso anterior, por ejemplo, se podría declarar un vector de punteros, y cada uno de esos vectores inicializarlo como un vector de enteros:

```
int *w[filas];
for (int i=0; i < filas; i++) {
    w[i] = malloc(cols*sizeof(int));
}
```

En este código, `w` es un vector de `filas` punteros. Cada uno de estos punteros contiene la dirección de memoria de una porción de memoria dinámica de tamaño `cols*sizeof(int)`.

Al pasar este vector por parámetro a una función, se lo hará de la siguiente manera:

```
void imprimir_matriz(int** v, size_t filas, size_t cols) {
```

Utilizando este formato será posible seguir accediendo a los datos contenidos en la matriz como `v[fila][columna]`.

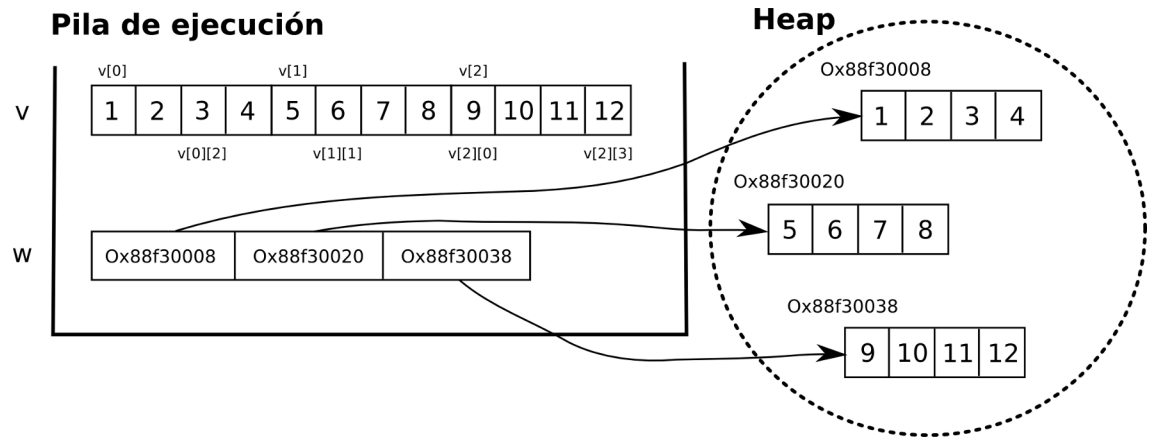

Figura 11.2: Ubicación de los vectores mostrados en la memoria

18. Cadenas y vectores de cadenas

Como ya se ha mencionado, las cadenas en C son arreglos de caracteres, terminados por un caracter `\0`.

Si bien es posible declararlas como **`char`** `cadena[largo]`, lo más usual es declararlas como **`char`** `*cadena` y luego asignar una dirección de memoria apropiada para el dato que se quiera representar.

Cuando se declara una cadena en memoria estática, como en el siguiente ejemplo:

```
char *cadena = "Algoritmos";
```

La variable `cadena` contiene una dirección de memoria, en la cual comienza el arreglo de caracteres, que tiene 11 caracteres (10 letras y un `\0` al final).

Si se quiere tener un arreglo que contenga varias cadenas, se deberá operar de manera similar a la mostrada para las matrices.

```
char* palabras[] = {"Hola", "que", "tal"};
```

Es decir que en este caso la variable `palabras` contiene un arreglo de tres punteros, cada uno de los cuales apunta a una porción de memoria donde están almacenadas las palabras.

Parámetros por línea de comandos

Se le llaman parámetros de línea de comandos a todo lo que se escriba después del nombre del programa en la invocación de un comando. Por ejemplo, hemos dicho que para compilar se usa una línea de la forma:

```
gcc hola.c -o hola
```

En este caso, estamos invocando al compilador `gcc` para que compile `hola.c` y genere el archivo ejecutable `hola`. Esto es posible, ya que el sistema operativo le pasa al gcc los parámetros que escribió el usuario al invocarlo, de forma que los parámetros que recibe gcc son:

```
"gcc", "hola.c", "-o", "hola"
```

En nuestro código, para recibir estos parámetros se utiliza, tradicionalmente, la *firma* de la función `main` que recibe parámetros:

```
int main(int argc, char *argv[]);
```

En este caso recibimos en `main` los parámetros de la invocación del programa. El primer parámetro será la cantidad de parámetros que se reciben y el segundo parámetro en un vector de punteros a caracteres.

Como se vio anteriormente, cada puntero a caracteres, es la dirección de memoria de un bloque de caracteres.

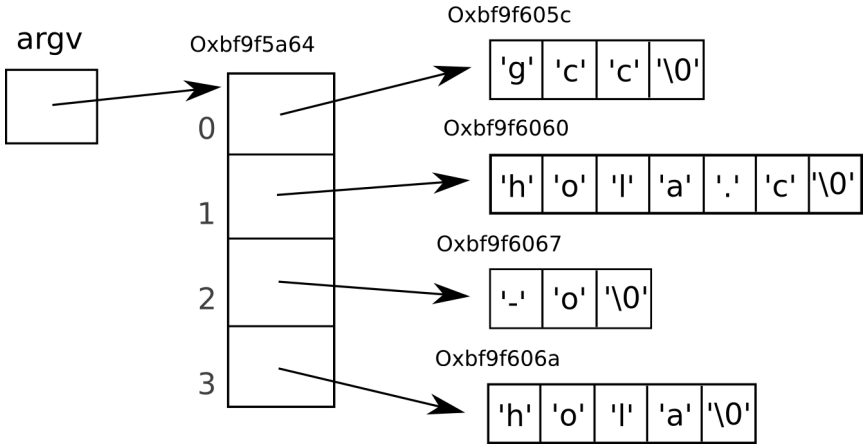

Figura 11.3: Estructura de los parámetros recibidos por la línea de comandos al compilar

Notar que la primera cadena apuntada por `argv`, es el nombre del comando que se llamó.

19. Ejemplo de recibir parámetros por línea de comandos

En UNIX existe un comando llamado **echo** cuya única función es imprimir todos los parámetros que se reciben por línea de comandos. Cada parámetro se imprime con un espacio entre parámetro y parámetro. Este sencillo programa puede escribirse en C de la siguiente forma.

```
1 #include <stdio.h>
2 int main(int argc, char *argv[])
3 {
4     if (argc > 1) {
5         printf("%s", argv[1]);
6     }
7     for (int i=2; i<argc; i++) {
8         printf(" %s", argv[i]);
9     }
10    printf("\n");
11    return 0;
12 }
```

En este código imprimimos todos los parámetros recibidos, omitiendo el nombre del comando (`argv[0]`). Una vez compilado se lo puede invocar de la siguiente forma:

```
./echo primer_parámetro  segundo_parámetro   etc
```

Y sin importar la cantidad de espacios entre un parámetro y otro el resultado será:

```
primer_parámetro segundo_parámetro etc
```

Ordenamiento recursivo por mezcla, *merge sort*

El método de ordenamiento *merge sort* es uno de los métodos de ordenamiento recursivos, basados en la técnica de dividir y conquistar. Se lo puede utilizar para ordenar cualquier estructura secuencial (vectores, listas, etc).

Los pasos de este método de ordenamiento son:

1. Cuando la longitud del vector sea 0 o 1, se considera que ya se encuentra ordenado. De no ser así:
2. Se divide el vector en dos partes de aproximadamente la mitad del tamaño.
3. Se ordena cada una de esas partes, utilizando este mismo método.
4. Tomando las dos partes ordenadas, se las intercala de forma ordenada, para obtener el vector original ordenado.

Por ejemplo, si el vector original es `[6, 7, -1, 0, 5, 2, 3, 8]`, se lo dividirá en dos partes: `[6, 7, -1, 0]` y `[5, 2, 3, 8]`, que serán ordenadas de forma recursiva. Luego de ordenarlas, se obtendrá: `[-1, 0, 6, 7]` y `[2, 3, 5, 8]`. Al intercalar ordenadamente los dos vectores ordenados se obtendrá la solución buscada: `[-1, 0, 2, 3, 5, 6, 7, 8]`.

20. Implementación básica

Será necesario programar dos funciones. Por un lado la función `merge_sort`, que será la función recursiva encargada de dividir la lista en dos hasta llegar a la condición de corte (cuando la lista tenga un tamaño menor que 2).

```
1 void merge_sort(int vector[], int inicio, int fin)
2 {
3     int largo = fin - inicio;
4     if (largo < 2) {
5         return;
6     }
7
8     int medio = inicio + (largo / 2);
9     merge_sort (vector, inicio, medio);
10    merge_sort (vector, medio, fin);
11    merge (vector, inicio, medio, fin);
12 }
```

Se puede ver que esta es una función extremadamente sencilla, cuya única tarea es dividir el vector en dos partes. Por otro lado, será necesario programar la función `merge`, que será la encargada de intercalar las partes una vez que estén ordenadas.

```
1 void merge (int vector[], int inicio, int medio, int fin)
2 {
3     int pos_1 = inicio;
4     int pos_2 = medio;
5     int aux[fin - inicio];
6     int pos_a = 0;
7
8     // Intercala ordenadamente
9     while ( (pos_1 < medio) && (pos_2 < fin) ) {
10        if ( vector[pos_1] <= vector[pos_2] ) {
11            aux[pos_a] = vector[pos_1];
12            pos_a++; pos_1++;
13        } else {
14            aux[pos_a] = vector[pos_2];
15            pos_a++; pos_2++;
16        }
17    }
18    // Copia lo que haya quedado al final del primer vector
19    while (pos_1 < medio) {
20        aux[pos_a] = vector[pos_1];
21        pos_a++; pos_1++;
22    }
23    // Copia lo que haya quedado al final del segundo vector
24    while (pos_2 < fin) {
25        aux[pos_a] = vector[pos_2];
26        pos_a++; pos_2++;
27    }
28
29    // Copia los valores del vector auxiliar al original
30    int a = 0;
31    int i = inicio;
32    while (i < fin) {
33        vector[i] = aux[a];
34        i++; a++;
35    }
36 }
```

Si bien esta función tiene unas cuantas líneas de código, su tarea no es muy compleja, simplemente inserta en un vector auxiliar los elementos del vector que ya se encuentran ordenados, de forma que sólo se recorren los elementos una sola vez.

21. Análisis de complejidad

Sea N la longitud del vector. Se pueden hacer las siguientes observaciones:

- En la función `merge` se ve que el tiempo que se tarda en intercalar dos vectores de longitud $N/2$ es proporcional a N, ya que todos los elementos se copian una vez al vector auxiliar y luego se los vuelve a copiar al vector original. Es posible, entonces, utilizar $A * N$ para representar ese tiempo.
- Si se denomina $T(N)$ al tiempo que tarda el algoritmo en ordenar un vector de longitud N, en la función `merge_sort` se puede ver que $T(N) = 2 * T(N/2) + A * N$, ya que la función simplemente se llama a sí misma con dos partes de la mitad de tamaño, y luego a la función `merge` con el tamaño total.
- Además, también en `merge_sort` se puede ver que el tiempo necesario para un vector de longitud menor a 2 es sólo el necesario en hacer una comparación. Es decir que, $T(1) = T(0) = B$.

Estos datos forman una ecuación de recurrencia, para resolverla, se supondrá que $N = 2^k$, quedando las ecuaciones:

$$\begin{aligned} T(N) = T(2^k) &= 2 * T(2^{k-1}) + A * 2^k && (11.1)\\ T(1) &= B && (11.2) \end{aligned}$$

Es posible resolver estas ecuaciones utilizando el *método telescópico*.

$$\begin{aligned} T(2^k) &= 2 * T(2^{k-1}) + A * 2^k && (11.3)\\ &= 2 * (2 * T(2^{k-2}) + A * 2^{k-1}) + A * 2^k && (11.4)\\ &= 2^2 * T(2^{k-2}) + 2 * A(2^k) && (11.5)\\ &\vdots && (11.6)\\ &= 2^i * T(2^{k-i}) + i * A * 2^i && (11.7)\\ &\vdots && (11.8)\\ &= 2^k * T(1) + k * A * 2^k && (11.9)\\ &= 2^k * B + k * A * 2^k && (11.10) \end{aligned}$$

Como $N = 2^k$ entonces $k = \log_2 N$, y por lo que esta resolución demuestra que $T(N) = B * N + A * N * \log_2 N$.

Como $A * N * \log_2 N$ es un término de mayor orden que $B * N$, el orden de este algoritmo es $O(N * \log_2 N)$.

Los valores de las constantes A y B son importantes a la hora de buscar la mejor implementación de *merge sort*, pero no para el cálculo del orden del algoritmo.

Por otro lado, al analizar el espacio que consume este algoritmo, se puede ver que para realizar el intercalado, necesita copiar el vector a un vectora auxiliar, es decir que duplica el espacio consumido.

22. Implementaciones más eficientes

Si bien el orden del algoritmo *merge sort* será siempre $O(N * \log_2 N)$, si se quiere una implementación realmente eficiente del ordenamiento, será necesario hacerle algunas mejoras a la implementación mostrada.

22.1. Implementación con un solo pedido de memoria

El valor A está asociado al tiempo necesario para ejecutar la función `merge`. Una de las operaciones que se puede eliminar de esta función es el pedido de memoria para el arreglo auxiliar, `int aux[fin-inicio];`, ya que esta operación consume tiempo en pedir la memoria (y luego devolverla, al terminar la función), que podría ahorrarse si se hiciera un único pedido de memoria para todo el algoritmo.

Para ello, será necesario crear una función adicional, que sea la que haga el pedido de memoria auxiliar, y luego llame a la función recursiva, con esa memoria ya reservada.

```
1 void merge_sort(int vector[], int largo)
2 {
3     int aux[largo];
4     msort(vector, 0, largo, aux);
5 }
```

La función recursiva es ahora `msort`, que es prácticamente igual a la vista previamente, siemplemente incluye el pasaje de la variable auxiliar.

```
1 void msort(int vector[], int inicio, int fin, int aux[])
2 {
3     int largo = fin - inicio;
4     if (largo == 1) {
5         aux[inicio] = vector[inicio];
6         return;
7     }
8
9     int medio = inicio + (largo / 2);
10    msort (vector, inicio, medio, aux);
11    msort (vector, medio, fin, aux);
12    merge (vector, inicio, medio, fin, aux);
13 }
```

Por otro lado, la función `merge`, ya no deberá realizar el pedido de memoria para alojar el vector adicional, sino que trabajará directamente sobre la misma porción del vector auxiliar que la utilizada para el vector de valores.

```
1 void merge (int vector[], int inicio, int medio, int fin, int aux[])
2 {
3     int pos_1 = inicio;
4     int pos_2 = medio;
5     int pos_a = inicio;
6
7     // Intercala ordenadamente (...)
8     // Copia lo que haya quedado al final del primer vector (...)
9     // Copia lo que haya quedado al final del segundo vector (...)
10
11    // Copia los valores del vector auxiliar al original
12    int i;
13    for (i = inicio; i < fin; i++) {
14        vector[i] = aux[i];
```

```
15          }
16  }
```

De esta forma se logró evitar tener que estar pidiendo memoria para el vector auxiliar una y otra vez. Sin embargo, esto no alcanza para decir que se cuenta con una versión realmente eficiente de *merge sort*.

22.2. Otras mejoras

Se puede seguir trabajando sobre el mismo algoritmo para agregarle varias otras mejoras, como por ejemplo:

Implementación sin copia inútil de los datos Otra operación que consume tiempo inútilmente es volver a copiar los datos del vector auxiliar al principal al terminar la función `merge`.

Esta copia puede evitarse si se opera alternadamente con el vector auxiliar y con el principal, de modo que el vector auxiliar de una llamada a `msort` es el principal de la llamada recursiva realizada dentro de la función, y así.

Uso de otros tipos de datos En los ejemplos mostrados se han usado vectores de enteros para hacer más simple el ejemplo, pero de la misma forma puede usarse cualquier otro tipo de dato que tengamos alguna forma de compararlo. O bien hacer una implementación que no le importe el tipo de dato con el que opera, y use una función que recibe por parámetro para comparar elementos.

Ordenamiento rápido, *quick sort*

El método de ordenamiento *quick sort* es el más famoso de los métodos de ordenamiento recursivos, su fama se basa en que puede ser implementado de forma muy eficiente y en la gran mayoría de los casos tiene el mismo orden de complejidad que *merge sort*. Al igual que este último, está basado en la técnica de dividir y conquistar.

Los pasos de este método de ordenamiento son:

1. Cuando la longitud del vector sea 0 o 1, se considera que ya se encuentra ordenado. De no ser así:
2. Se elige un elemento del vector como *pivote*. Generalmente será el primero o el último.
3. Se reordenan los elementos del vector de modo que quede dividido en tres partes (**partición**): los elementos menores al pivote, el pivote y los elementos mayores al pivote. Al terminar este paso, el pivote queda en su lugar definitivo.
4. Se repite el mismo proceso para cada una de las partes que no contienen al pivote (los menores y los mayores).

Por ejemplo, si el vector original es `[6, 7, -1, 0, 5, 2, 3, 8]` y se elige el primer elemento como pivote (`6`), la partición del vector será: `[-1, 0, 5, 2, 3]`, `6`, `[7, 8]`. Se procederá a ordenar recursivamente `[-1, 0, 5, 2, 3]` y `[7, 8]`, de modo que el vector final será `[-1, 0, 2, 3, 5, 6, 7, 8]`.

23. Implementación básica

En este caso se implementará una función `quick_sort`, que se encargará tanto de realizar la partición, como de llamarse recursivamente hasta que no haya más elementos que ordenar.

La elección del pivote depende de cada implementación de *quick sort*, en este caso se elige el primer elemento del vector como pivote.

```
1 void quick_sort(int vector[], int inicio, int fin)
2 {
3     int pivote = inicio;
4     int ult_menor = inicio;
5 
6     if ( (fin - inicio) < 2 ) {
7         return;
8     }
9 
```

```
10      int i;
11      for (i = pivote + 1; i < fin; i++) {
12          if ( vector[i] < vector[pivote] ) {
13              ult_menor++;
14              swap(vector, i, ult_menor);
15          }
16      }
17
18      // Coloca el pivote al final de los menores y el último
19      // menor en el primer lugar.
20      swap(vector, pivote, ult_menor);
21      // Ordena cada una de las mitades
22      quick_sort(vector, inicio, ult_menor);
23      quick_sort(vector, ult_menor+1, fin);
24
25  }
```

El bucle principal de la función recorre los elementos del vector una única vez, cambiando de lugar aquellos que son menores al pivote para que queden en la primera parte y que los mayores queden en la segunda.

Una vez terminado este bucle, se coloca el pivote en el medio de ambas partes, de modo que quede ubicado en su posición final.

La función `swap` utilizada en esta porción de código, recibe un vector y dos posiciones dentro del vector, e intercambia los valores que se encuentran en esas dos posiciones:

```
1  void swap(int vector[], int pos_1, int pos_2)
2  {
3      int aux = vector[pos_1];
4      vector[pos_1] = vector[pos_2];
5      vector[pos_2] = aux;
6  }
```

Esta función puede utilizarse siempre que se necesite intercambiar dos elementos de un vector.

24. Análisis de complejidad

A simple vista, el algoritmo de *quick sort* puede parecer muy similar al de *merge sort*, ya que en ambos casos se divide a la lista en dos, y se opera sobre partes cada vez más pequeñas.

Sin embargo, algo importante a tener en cuenta es que en el caso del *quick sort* el orden que tenga el algoritmo dependerá en una gran parte de la elección del pivote, ya que no es lo mismo elegir un valor que se encuentre aproximadamente en el medio, de forma que las dos partes sean aproximadamente del mismo tamaño, que elegir un valor que se encuentre en uno de los extremos, de modo que una de las partes mida mucho más que la otra.

Asumiendo que el valor elegido se encuentra aproximadamente en el medio, se puede ver que el tiempo requerido para ejecutar el algoritmo es:

$$T(N) = A * N + 2 * T(N/2) \tag{11.11}$$

$$T(1) = B \tag{11.12}$$

Donde B es el tiempo requerido por el caso base, y A es el valor asociado a recorrer el vector y cambiar los elementos de lugar en el bucle principal. Puede verse que estas ecuaciones son las mismas que para *merge sort*.

Sin embargo, cuando el pivote elegido no divide ambas partes al medio, el comportamiento no es tan bueno. En el peor caso (cuando una parte queda con todos los elementos menos el pivote y la otra vacía), será:

$$T(N) = A * N + T(N-1) \tag{11.13}$$
$$T(1) = B \tag{11.14}$$

De modo que aplicando el método telescópico, similar al utilizado anteriormente:

$$\begin{aligned} T(N) &= A * N + T(N-1) && (11.15)\\ &= A * N + A * (N-1) + T(N-2) && (11.16)\\ &= A * (N + N - 1 + N - 2) + T(N-3) && (11.17)\\ &\vdots && (11.18)\\ &= A * (N + N - 1 + N - 2 + \ldots + 2) + B && (11.19)\\ &= A * \sum_{i=2}^{N} i + B && (11.20)\\ &= A * \frac{N^2 + N}{2} - 1 + B && (11.21) \end{aligned}$$

Se puede ver que en el peor caso, el orden será $O(N^2)$, mucho peor que el $O(Nlog_2N)$ visto anteriormente. Sin embargo, se pueden tomar recaudos especiales para que este peor caso sea extremadamente improbable, y que en la práctica se pueda considerar que el algoritmo se comporta como $O(Nlog_2N)$.

25. Implementaciones más eficientes

25.1. Elección del pivote

Si se elige el primer elemento (o el último), el algoritmo resulta muy inconveniente para el caso de una lista que ya se encuentra ordenada, y este es un caso que en ciertas situaciones es esperable que suceda.

Es por eso que una optimización sencilla es intercambiar el elemento del medio con el que se vaya a utilizar de pivote antes de comenzar el bucle principal.

```
1     int medio = (fin + inicio) / 2;
2     swap(vector, pivote, medio);
```

Otras técnicas de elección del pivote incluyen:

- Elegir un elemento aleatoriamente, esto hace que en promedio sea mucho más probable tener un buen caso que uno malo, pero no elimina la posibilidad del peor caso.

- Recorrer la lista y buscar el elemento que ocupará la posición central de la lista. Eso asegura que el orden sea siempre $O(Nlog_2N)$, pero decrementa mucho la eficiencia del caso base.
- Elegir tres elementos de la lista (por ejemplo, el primero, el del medio y el último), y quedarse con el del medio de los tres como pivote.

25.2. Reducción de la cantidad de intercambios

En la implementación vista, puede suceder que se hagan numerosos intercambios innecesarios, cuando un elemento ya es menor que el pivote, y simplemente haría falta avanzar la variable que indica la posición del último menor.

Una implementación alternativa de *quick sort* se basa en esta idea para tratar de minimizar la cantidad de intercambios. Se cuenta con dos variables, que se utilizan para saltear los elementos que no hace falta cambiar de lugar, y sólo cambiar aquellos que es necesario.

```
1 void quick_sort(int vector[], int inicio, int fin)
2 {
3     if ( (fin - inicio) < 2 ) {
4         return;
5     }
6     int izq = inicio + 1;
7     int der = fin - 1;
8     int pivote = inicio;
9
10    // Cambia el del medio con el primero.
11    // (optimización para vectores ordenados).
12    int medio = (izq + der) / 2;
13    swap(vector, pivote, medio);
14
15    while (izq <= der) {
16        while ( (izq <= der) && (vector[der] >= vector[pivote]) )
17            der--;
18        while ( (izq <= der) && (vector[izq] < vector[pivote]) )
19            izq++;
20        if ( izq < der )
21            swap(vector, izq, der);
22    }
23
24    // Coloca el pivote al final de los menores y el último
25    // menor en el primer lugar.
26    swap(vector, pivote, der);
27    // Ordena cada una de las mitades
28    quick_sort(vector, inicio, der);
29    quick_sort(vector, der+1, fin);
30 }
```

Como se puede ver, se ha reemplazado el bucle principal, por otro que recorre el vector desde ambas puntas hacia el medio, buscando los elementos que necesitan ser intercambiados.

25.3. Utilización de otros algoritmos

En particular para los casos de las partes más pequeñas, al tener menos elementos, sin importar cuál se elija como pivote, es más probable que se asemejen al peor caso.

Es por ello que una técnica de optimización puede incluir utilizar un algoritmo alternativo, como por ejemplo el de ordenamiento por inserción, para secuencias de pocos elementos.

Por otro lado, puede también implementarse un contador que verifique la profundidad de la recursión y cuando esta exceda el nivel esperado por el algoritmo, pasar a utilizar otro algoritmo de ordenamiento, como *merge sort* o *heap sort*.

26. Quick sort en la biblioteca estándar de C

Entre las funciones que provee la biblioteca estándar de C, se incluye una implementación de quick sort. Dado que la biblioteca estándar está altamente probada y seguramente contenga optimizaciones avanzadas, es en general una buena idea usar las funciones que provistas antes que usar las propias.

En el caso del *quick sort*, la función se llama `qsort` y se encuentra definida en el encabezado `<stdlib.h>`, su prototipo es:

```
1 void qsort(void *base, size_t nmemb, size_t size,
2            int(*compar)(const void *, const void *));
```

Que puede ser intimidante por la cantidad y complejidad de parámetros que recibe, en gran parte debido a la generalidad del código. `base` se refiere al vector a ordenar, `nmemb` es la cantidad de elementos del vector, `size` es el tamaño en bytes de un elemento, `compar` es la función que se debe usar para comparar.

Este tipo de funciones se verán con más detalle más adelante.

Compilación de varios archivos en C

En todo programa es importante modularizar el código de forma que se facilite la reutilización y se minimice la repetición de código. En particular, cuando se trata de tipos abstractos de datos, es importante tener un módulo correspondiente a cada tipo abstracto.

Por otro lado, para que un tipo abstracto de datos sea realmente *abstracto* es recomendable que la implementación de las operaciones correspondientes al tipo estén separadas de los prototipos de estas operaciones, de modo que quien las utiliza se concentre únicamente en cuáles son las operaciones y no en cómo se llevan a cabo.

27. Encabezado, implementación y código objeto

En C, esto se logra mediante la separación de cada módulo en un archivo `.h`, que contiene las declaraciones de estructuras, constantes y prototipos de las funciones, que es llamado el *encabezado* del módulo, y un archivo `.c` que contiene las implementaciones correspondientes.

Cada uno de los archivos `.c` se utiliza para generar un archivo `.o` que contiene el *código objeto*, es decir el código de máquina, correspondiente a cada módulo.

Cuando se compila un programa completo, todos los `.o` que se hayan generado a partir de los módulos programados deben combinarse en un sólo ejecutable.

27.1. Inclusión de otros encabezados

Cuando un módulo requiere de funciones definidas en otros módulos, debe incluir los encabezados (los archivos `.h`) en los que esas funciones están definidas. Esta inclusión se realiza normalmente dentro del encabezado correspondiente al módulo en cuestión.

Es posible que al momento de construir un programa de tamaño considerable suceda que hay varios módulos que dependen de otro. De modo que podría suceder que este otro se incluya varias veces, lo cual es problemático y debe ser evitado.

Para solucionar este problema, se utilizan las construcciones condicionales del preprocesador, haciendo que el código definido dentro de un encabezado se incluya en el programa un única vez. Por ejemplo:

```
#ifndef __ENUM_H
    #define __ENUM_H
    typedef enum {OK, ERROR} estado;
    typedef enum {HUMANO, COMPUTADORA} jugador_t;
#endif
```

En este caso, el preprocesador verifica si está definida la variable `__ENUM_H`. De no estar definida, la define y luego define los tipos enumerados correspondientes a este encabezado.

En cambio, si la variable ya estaba definida, significa que este encabezado ya fue procesado, con lo cual no se hace nada.

28. Compilación con make

Cuando los módulos que componen un programa son muchos, los pasos necesarios para compilarlo pueden ser muchos y tener que regenerarlos manualmente cada vez que se los modificque sería una tarea demasiado tediosa. Es por ello que existe una herramienta llamada `make`, encargada de realizar todos los pasos de compilación y necesarios y de hacerlos sólo cuando haga falta.

Esta herramienta utiliza, a modo de configuración, los archivos `Makefile` en donde se indican los pasos a realizar para la compilación tanto de los módulos como del programa principal.

En estos archivos, básicamente, se pueden definir variables y reglas para compilar los distintos módulos.

28.1. Un `Makefile` sencillo

A continuación un ejemplo de cómo puede verse un posible archivo `Makefile`.

```
CFLAGS = -g -Wall -std=c99
EXEC = miprog
OBJ = lista.o pila.o
CC = gcc

all: $(EXEC)

lista.o: lista.c lista.h
	$(CC) $(CFLAGS) -c lista.c

pila.o: pila.c pila.h
	$(CC) $(CFLAGS) -c pila.c

$(EXEC): $(OBJ) miprog.c
	$(CC) $(CFLAGS) $(OBJ) miprog.c -o $(EXEC)
```

28.2. Variables

En la primera parte se declaran 4 variables, `CFLAGS` son los *flags* (parámetros) de compilación utilizados, en este caso se trata del parámetro que incluye la información para depuración `-g` y el parámetro para que advierta sobre todos los posibles problemas que el compilador encuentre, `-Wall`.

Luego se declara el nombre que tendrá el programa ejecutable. En este caso no tiene extensión, puesto que es un programa para sistemas UNIX (Linux, BSD, Solaris, etc). Si se estuviera compilando para un sistema Windows, la variable `EXEC` sería `miprog.exe`.

Luego se listan cuáles serán los módulos que deberán transformarse en código objeto, y finalmente se coloca el nombre del compilador. Separar la información de esta manera permite

que si es necesario hacer un cambio en la forma de compilar un programa, el trabajo para realizarlo sea mínimo.

Es importante notar que en el `Makefile`, las variables se definen simplemente con `VARIABLE=VALOR`, pero luego para utilizarlas, se lo hace de la forma `$(VARIABLE)`.

28.3. Reglas de compilación

Luego de las variables, se definen cada uno de los archivos a generar, los archivos de los cuales estos dependen, y las acciones a llevar a cabo para compilar los archivos correspondientes.

Cuando se ejecuta el comando `make` sin ningún parámetro, se ejecuta automáticamente la primera de todas las reglas, es por eso que esta regla normalmente se llama `all` y simplemente indica cuál es el archivo a generar.

Luego de esta regla especial, se encuentran las reglas de compilación de cada uno de los módulos del programa. Las reglas tienen un formato específico que se debe cumplir para que funcione el `Makefile`.

```
archivo: dependencias
        acciones
```

Esto significa que `archivo` debe generarse cada vez que cambie una de las `dependencias`, ejecutando las `acciones`.

Es importante notar que para que el `make` funcione correctamente, las acciones a realizar deben tener un tabulador de separación desde el comienzo de línea. Puede haber más de una acción, de a una por línea, siempre que se mantenga un tabulador de separación.

En particular, la regla que se encarga de generar el programa principal es diferente a las otras, ya que incluye una mayor cantidad de dependencias y de variables.

```
$(EXEC): $(OBJ) miprog.c
        $(CC) $(CFLAGS) $(OBJ) miprog.c -o $(EXEC)
```

Esta regla indica que el archivo indicado mediante la variable `$(EXEC)` definida previamente debe generarse cuando cambie cualquiera de los archivos objeto, ya que si estos cambian, también debe cambiar el ejecutable final, o bien si cambia el código principal del programa.

Es importante notar que mediante estas reglas, `make` no sólo es capaz de compilar los archivos de forma correcta, sino que también es capaz de realizarlo sólo cuando sea necesario. Para ello, verifica que la fecha de modificación de los archivos listados como dependencias sea anterior al archivo a generar. De no ser así, vuelve a generarlo, ya que algo ha cambiado.

28.4. Reglas genéricas

Cuando la gran mayoría de los archivos del programa se compilan de una misma forma, es posible simplificar el archivo `Makefile`, mediante el uso de reglas genéricas.

Por ejemplo, para el caso del `Makefile` visto anteriormente, las dos líneas que generan los archivos `.o` podrían simplificarse en una sola de la siguiente forma:

```
%.o: %.c %.h
        $(CC) $(CFLAGS) -c $<
```

Esta regla significa que para generar un archivo `.o` es necesario contar con un archivo del mismo nombre `.c` y otro del mismo nombre `.h`. En la regla de compilación se utiliza la variable especial `$<`, que toma el valor de la primera de las dependencias listadas (el archivo `.c`.). Existe también `$@`, que toma el nombre del archivo que está siendo generado en esa regla.

28.5. Acciones comunes

Además de compilar, es común querer eliminar los archivos generados, de modo que queden únicamente los archivos fuente del programa. Esta acción normalmente se realiza mediante una regla especial llamada `clean`.

Si bien puede llevar cualquier nombre, lo más usual es ponerle este nombre ya que tanta gente la llama así que cualquier programador que se encuentre con un `Makefile` esperará encontrar una regla con este nombre que realice esta acción.

```
clean:
        rm $(OBJ) $(EXEC)
```

Para utilizar esta regla (o cualquier otra que no sea la predeterminada), debe invocarse el comando `make` con el nombre de la regla como parámetro. Es decir, `make clean`.

Al igual que con `clean`, existen otras acciones comunes que se suelen incluir en la mayoría de los programas. Las más conocidas:

build Para compilar el código (equivalente a la regla `all` del ejemplo mostrado).

install Para instalar el código compilado en el sistema.

uninstall Para desinstalar el programa que haya sido instalado.

28.6. Variables comunes

Así como existen reglas comunes, que la mayoría de los programadores están acostumbrados a encontrar en los archivos `Makefile`, también existen variables que suelen estar presentes.

CFLAGS Que estaba en el ejemplo mostrado, son los parámetros pasados al compilador.

LDFLAGS Son los parámetros pasados al enlazador.

PREFIX Utilizado cuando hay una regla de instalación, es el directorio a partir del cual se instalarán los archivos.

28.7. Reglas, archivos y PHONY

Cada regla de makefile tiene como finalidad crear un archivo, sin embargo hay reglas como la regla `clean` que mostramos más arriba que no genera ningún archivo, tampoco depende de ningún archivo (de hecho borra archivos, pero esto es parte de la acción y a make no le importa que hace la acción).

Es más, si creamos un archivo llamado `clean` make creerá que la regla está satisfecha. Para evitar que `make` revise archivos que no vamos a generar es de bastante recomendable declarar las reglas que no generan archivos como `.PHONY`, por ejemplo:

```
.PHONY: install clean
```

Archivos en C

En este apunte se verá una referencia de las funciones y conceptos de archivos usado en C, resaltando algunas peculiaridades que no se ven en otros lenguajes. Pero de ninguna manera pretende ser un apunte completo sobre el uso de archivos en general y se asume cierta experiencia al respecto.

Una de las peculiaridades de C es que, todos los programas al ejecutarse ya tienen tres archivos abiertos, estos son: la entrada estándar (*stdin*), salida estándar (*stdout*) y salida de error (*stderr*). Los primeros dos son los que usan las funciones de entrada y salida del usuario, como `scanf` y `printf`, respectivamente. El tercero es un archivo de salida destinado al envío de errores de ejecución y por omisión saldrán en la misma salida que los de salida externa.

Siendo que lo que tiene que ver con archivos es normalmente entrada o salida del programa, las funciones listadas en este apunte están declaradas en el encabezado `<stdio.h>`.

29. Entrada y salida de una terminal

La entrada y salida de una terminal en C se comporta de una forma similar a la lectura y escritura de archivos, por lo que se listan a continuación algunas de las funciones de entrada y de salida.

Tanto la entrada como la salida estándar suelen tener un *buffer*, es decir una memoria intermedia, en este caso por líneas, por lo que al intentar leer de entrada estándar mediante `scanf`, el programa se quedará esperando hasta que se termine una línea en la entrada, aún si sólo se quiere leer un caracter. El resto de línea no procesada será la entrada de las siguientes llamadas a las funciones de entrada.

Algo similar sucede con la salida por consola, cuando se utiliza `printf`, la salida suele tener también un *buffer* orientado a líneas, por lo que hasta que no se termine una línea, la salida no se emite en la terminal.

29.1. Manejo de caracteres de a uno

Sin embargo, no es la única opción. Existen otras funciones como:

```
int getchar(void);
```

Esta función permite leer un único caracter desde la entrada estándar, devuelve el valor del caracter leído o, en caso de haberse terminado la entrada, el valor especial `EOF`.

De la misma forma, para emitir un único caracter por la terminal:

```
int putchar(int c);
```

Esta función permite escribir un caracter en la terminal, devuelve el valor del caracter escrito o bien `EOF` en caso de error.

30. Abrir archivos

Para abrir un archivo en C se utiliza la función `fopen`[18], cuyo prototipo es:

```
FILE *fopen(const char *ruta, const char *modo);
```

El primer parámetro es el nombre del archivo, y el segundo el modo de apertura, que puede ser:

- `r` Sólo lectura, se posiciona al principio del archivo.
- `r+` Lectura y escritura, se posiciona al principio del archivo.
- `w` Borra el contenido del archivo o crea uno nuevo, sólo escritura, se posiciona al principio del archivo.
- `w+` Borra el contenido del archivo o crea uno nuevo, lectura y escritura, se posiciona al principio del archivo.
- `a` Abre para añadir (escribir al final del archivo). El archivo se crea si no existe. Se posiciona al final del archivo.
- `a+` Abre para leer y añadir (escribir al final del archivo). El archivo se crea si no existe. Se posiciona al final del archivo.

Además, el archivo puede abrirse en modo *archivo de texto* (por omisión) o en modo *archivo binario* (agregándole una `b` al modo). Los archivos de texto tienen un tratamiento especial para el caracter fin de línea, mientras que con los archivos binarios se accede a los datos en crudo.

El valor devuelto por `fopen` es un puntero de tipo `FILE` que representa a los archivos en la biblioteca estándar. En caso de error, el valor devuelto es `NULL`.

31. Cerrar archivos

Cerrar un archivo es más sencillo:

```
int fclose(FILE *archivo);
```

Devuelve `0` si tuvo exito, o `EOF` en caso de error.

32. Leer o escribir de un archivo

De la misma manera que `getchar` para leer un caracter de la entrada estándar, existe `fgetc` [19] para leer un único caracter de un archivo.

```
int fgetc(FILE *archivo);
```

De hecho, la siguiente función es prácticamente equivalente a la función `getchar()`.

```
int mi_getchar(void)
{
    return fgetc(stdin);
}
```

De la misma forma, existen `fputc`, para escribir un caracter a un archivo, `fscanf`, para leer con formato de un archivo, `fprintf` un archivo.[20], para escribir con formato a Sus prototipos son:

[18] Para más información: `man 3 fopen`.
[19] Para más información: `man 3 fgetc`.
[20] Para más información: `man 3 fputc`, `man 3 fscanf`, `man 3 fprintf`.

```
int fputc(int c, FILE *archivo);
int fscanf(FILE *archivo, const char *formato, ...);
int fprintf(FILE *archivo, const char *formato, ...);
```

Además de estas funciones existen:

```
char *fgets(char *buffer, int tamanio, FILE *archivo);
int fputs(const char *buffer, FILE *archivo);
```

La función `fgets` lee el archivo hasta encontrar un fin de línea, un fin de archivo o haber llegado a leer `tamanio` bytes. Cuando lee un fin de línea lo deja en el `buffer`. Devuelve la dirección del `buffer` o bien `EOF` si se trata de leer estando al final del archivo.

La función `fputs` escribe la cadena apuntada por `buffer` en `archivo`. Devuelve la cantidad de bytes escritos o bien `EOF` en caso de error.

Las funciones `fgets` y `fputs` constituyen la forma estándar de leer o escribir líneas en un archivo, si bien puede suceder que lo que se lea no sea una línea completa (cuando la línea ocupa más espacio que `tamanio`.

Si bien tienen un paralelo que trabaja sobre la entrada y salida estándar, esas funciones no se utilizan ya que pueden dar lugar a varios problemas de seguridad.

33. Otras funciones de archivos

Otras funciones que vale la pena mencionar son:

```
int fflush(FILE *archivo);
int feof(FILE *archivo);
```

La función `fflush` fuerza la escritura de los buffers que estén pendientes en el archivo. Devuelve 0 si se ejecutó correctamente, o `EOF` en caso de error. Puede utilizarse para evitar el comportamiento del buffer por líneas de las salida estándar.

La función `feof` devuelve algo distinto de cero si se encuentra al final del archivo o `0` en caso contrario.

34. Archivos binarios

Los archivos de texto son sencillos de procesar y faciles de leer aún fuera del programa que los usa, el éxito en los últimos años de los formatos XML, HTML, SVG, etc, demuestra su gran flexibilidad. Por otro lado, los archivos binarios permiten almacenar la información de forma que sea muy eficiente acceder a ella.

El formato a utilizar en una aplicación se debe decidir según el uso que se le vaya a dar a los archivos, si se quiere que sean legibles por seres humanos, si se quiere poder compartir la información entre aplicaciones, o si simplemente se quiere poder leer y guardar la información de la forma más eficiente.

Las funciones vistas hasta ahora son las más utilizadas al trabajar sobre archivos de texto, estas pueden servir para archivos binarios, pero además se necesitarán las siguientes:

```
size_t fread(void *buffer, size_t tamanio, size_t cantidad, FILE *archivo);
size_t fwrite(const void *buffer, size_t tamanio, size_t cantidad,
              FILE *archivo);
```

La función `fread` lee `cantidad` bloques de bytes de `tamanio` bytes cada uno, de un archivo, almacenandolos en `buffer`. Devuelve la cantidad de elementos leídos del archivo, en el caso de estar en el final del archivo devolverá 0.

De la misma forma `fwrite` escribe `cantidad` bloques de bytes de `tamanio` bytes cada uno en archivo y devuelve la cantidad de elementos escritos.

```
int fseek(FILE *archivo, long desplazamiento, int origen);
```

Se mueve dentro el archivo, `desplazamiento` es un valor relativo a `origen`, puede referirse al principio del archivo (`SEEK_SET`), a la posición actual (`SEEK_CUR`) o al final del archivo (`SEEK_END`). El valor devuelto será 0 en caso de exito o `-1` en caso de error.

```
long ftell(FILE *archivo);
```

Devuelve la posición actual del archivo, o `-1` en caso de error.

35. Ejemplo: Copiar un archivo

En este ejemplo vemos el uso de varias de las funciones mencionadas anteriormente. El código copia un archivo de un forma muy ineficiente, leyendolo de 1 caracter. Se muestra también el uso de `stderr`.

Podemos mejorarlo un poco leyendo por lineas en vez de caracter a caracter.

```
enum {MAXLINE = 1024};
...
    char buffer[MAXLINE], *aux;
    do {
        aux = fgets(buffer, MAXLINE, origen);
        if ( aux != NULL ) {
            fputs(buffer, destino);
        }
    } while (aux != NULL);
```

Se puede mejorar la eficiencia de este código utilizando `fread` y `fwrite`.

Código 1 `copiar.c`**:** Copia un archivo

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

int main(void)
{
    FILE *origen, *destino;
    int  valor;

    origen = fopen("copiar.c","r");
    if ( origen == NULL ) {
        fprintf(stderr, "Error al abrir el archivo origen");
        exit(1);
    }

    destino = fopen("copiar2.c","w");
    if ( destino == NULL ) {
        fprintf(stderr, "Error al abrir el archivo destino");
        exit(1);
    }

    do {
        valor = fgetc(origen);
        if ( valor != EOF ) {
            fputc(valor,destino);
        }
    } while (valor != EOF);
    fclose(origen);
    fclose(destino);

    return 0;
}
```

Licencia y Copyright