UNAM - Facultad de Ciencias
Redes de Computadoras 2019-2

Guía rápida de C

Por: Andrea Itzel González Vargas

7 de febrero de 2019

Índice

C es un lenguaje estructurado que fue diseñado por Dennis Ritchie en 1972, mientras trabajaba para *AT&T Bell Laboratories*. A pesar de su antigüedad, y menor accesibilidad de uso, comparado con lenguajes más recientes como Perl, Python o Java, C sigue siendo uno de los principales lenguajes de programación, debido a que permite programar en un nivel más bajo que otros lenguajes.

1. Compilación

Los archivos escritos en C se guardan con la extensión `.c`. Para compilar un archivo `.c` se hace uso de `gcc` de la siguiente manera:

```
$ gcc <archivo.c>
```

Algunas opciones que se pueden utilizar con `gcc` son: [1]

- `-o`: Indica el nombre del archivo ejecutable resultado de la compilación, por defecto es `a.out`.
- `-ggdb`: Produce información de depuración a ser usada por GDB. [2]
- `-w`: Deshabilita despliegue de advertencias.
- `-Wall`: Habilita despliegue de todas las advertencias sobre malas prácticas de programación que son fáciles de corregir.
- `-Q`: Imprime información de cada función al ser compilada e imprime estadísticas sobre éstas.
- `-c`: Compila o ensambla el código fuente sin entrar a la etapa de enlace. Genera archivos objeto con extensión `.o`.
- `-m32`: Indica que el programa resultante sea de 32 bits.
- `-fstack-protector`: Crea código extra para evitar desbordamientos de búfer (*buffer overflows*), está habilitada por defecto. Para deshabilitarla usar `-fno-stack-protector` (no recomendado).

1.1. make

`make` es una herramienta que permite compilar con mayor facilidad y eficiencia un programa. Funciona utilizando un archivo que debe de llamarse `Makefile` (o `makefile`). Un archivo `Makefile` consiste de reglas con la forma siguiente:

```
objetivo ...: [dependencias ...]
    comando
    ...
```

- Un `objetivo` es el nombre de un archivo generado por un programa (archivos binarios, archivos objeto...) o el nombre de una acción a llevar a cabo.
- Una `dependencia` es un archivo utilizado para crear el objetivo. Éstas son opcionales.

[1] Más información sobre `gcc`: `https://linux.die.net/man/1/gcc`
[2] `https://www.gnu.org/software/gdb/`

- Un `comando` es una acción que es llevada a cabo por `make`, un objetivo puede tener varios comandos, cada uno separado por un salto de línea y al principio deben de llevar un carácter de tabulador.

Un archivo objetivo es creado cuando alguna de las dependencias es modificada. Un ejemplo sencillo de un archivo `Makefile` es el siguiente:

Supongamos que se tiene los siguientes archivos:

definiciones.h:

```
#ifdef WIN32
#define TIPO_USUARIO "Windows"
#elif defined __unix__
#define TIPO_USUARIO "UNIX"
#else
#define TIPO_USUARIO "???"
#endif
void imprime();
```

imprime.c:

```
#include<stdio.h>
void imprime(char * mensaje) {
    puts(mensaje);
}
```

main.c:

```
#include<stdio.h>
#include"definiciones.h"
int main() {
    char mensaje[100];
    sprintf(mensaje, "Hola, estas utilizando %s", TIPO_USUARIO);
    imprime(mensaje);
}
```

Se crea entonces el siguiente archivo `Makefile`:

```
# Ejecuta el comando cuando se modifiquen main.o o imprime.o,
# creandose el archivo ejecutable mensaje
mensaje: main.o imprime.o
    gcc main.o imprime.o -o mensaje
# Ejecuta el comando cuando se modifiquen main.c o dependencias.h,
# creandose el archivo main.o
main.o: main.c dependencias.h
    gcc -c main.c
# Ejecuta el comando cuando se modifique imprime.c, creandose el
# archivo imprime.o
imprime.o: imprime.c
    gcc -c imprime.c
```

```
# Elimina los archivos objeto, asi como el ejecutable
clean:
    rm -f *.o mensaje
```

Para compilar los archivos se utiliza entonces el comando `make`:

```
[chepe@chepeman ejemplo]$ make
gcc -c main.c
gcc -c imprime.c
gcc main.o imprime.o -o mensaje
[chepe@chepeman ejemplo]$
```

Podemos ver que se crean los archivos objeto y el ejecutable `mensaje`:

```
[chepe@chepeman ejemplo]$ l
dependencias.h  imprime.c  imprime.o  main.c  main.o  Makefile  mensaje*
[chepe@chepeman ejemplo]$
```

Si se modifica el archivo `imprime.c`, entonces se corren únicamente las reglas con los objetivos `imprime.o` y `mensaje`:

```
[chepe@chepeman ejemplo]$ make
gcc -c imprime.c
gcc main.o imprime.o -o mensaje
[chepe@chepeman ejemplo]$
```

Al modificarse el archivo `definiciones.h`, se corren únicamente las reglas con los objetivos `main.o` y `mensaje`:

```
[chepe@chepeman ejemplo]$ make
gcc -c main.c
gcc main.o imprime.o -o mensaje
[chepe@chepeman ejemplo]$
```

Al correr `make clean` se ejecuta el comando de la regla con el objetivo `clean`:

```
[chepe@chepeman ejemplo]$ make clean
rm -f *.o mensaje
[chepe@chepeman ejemplo]$ l
dependencias.h  imprime.c  main.c  Makefile
[chepe@chepeman ejemplo]$
```

2. Estructura de un programa

Cada programa escrito en C consta de variables globales y funciones, siendo las variables globales aquellas que no hayan sido declaradas dentro de una función. La función principal dentro de un programa es la que tiene por nombre `main`, la cual es la primera en ejecutarse. A continuación se explica la estructura de las variables y las funciones.

3. Variables y tipos

3.1. Variables

En C las variables se declaran de manera estática, con el siguiente formato:

```
tipo nombre;
```

A una variable se le asigna un valor de la siguiente manera:

```
tipo nombre = expresion; //Si la variable no ha sido declarada previamente
nombre = expresion; //Si la variable ha sido declarada previamente
```

3.2. Tipos

Los tipos en C pueden o no tener signo. Una variable con signo utiliza el primer bit para indicar si su valor es positivo o negativo, así que si consta de N bits, tendra N - 1 bits para amacenar el valor. Una variable sin signo utiliza los N bits que tiene asignados para guardar su valor, el cual siempre se toma como positivo. La declaración de una variable sin signo se hace de la siguiente manera:

```
unsigned <tipo>.
```

Cada tipo cuenta a demás con un especificador de formato, el cual indica cómo se debe de imprimir una variable, cada especificador empieza con el signo '%'.

Existen los siguientes tipos:

Tipo	Tamaño (bytes)	Especificador de formato	Especificador de formato (unsigned)
`char`	1	`%c, %x, %o`	`%c, %u`
`short [int]`	2	`%hd, %x, %o`	`%hu, %u`
`int`	4	`%d, %x, %o`	`%u`
`long [int]`	8	`%ld, %lx, %lo`	`%lu`
`long long [int]`	8	`%lld, %llx, %llo`	`%llu`
`float`	4	`%f`	-
`double`	8	`%lf`	-
`long double`	16	`%Lf`	-
`void`	-	-	-

Los tipos `float`, `double`, `long float`, `long double` y `void` no pueden ser `unsigned`. El tamaño de los tipos puede variar dependiendo de la implementación del compilador, arquitectura de la máquina, opciones de compilador, etc.

`void` es un tipo especial que, como se verá más adelante, tiene un propósito dependiente del contexto.

Cabe destacar que C no tiene un tipo booleano como otros lenguajes, ésto es porque se hace uso del valor 0 para indicar un valor falso, cualquier otro valor se toma como verdadero.

Para darle un nuevo nombre o alias a un tipo dentro de un programa, se utiliza la palabra reservada `typedef`.

Ejemplo:

```
typedef unsigned char BYTE;
int main() {
    BYTE b = 8;
}
```

En este ejemplo se le está asignando al tipo `unsigned char` el alias `BYTE`.

3.3. Conversión de tipos

Una variable de un tipo puede utilizarse como si fuera de otro tipo, usando la conversión de tipos. Por ejemplo, si se tiene dos enteros y se quiere operar sobre ellos con una operación de tipo `float` en vez de tipo `int`, entonces se debe de tomar alguno de los valores como de tipo `float`:

```
#include<stdio.h>
int main() {
    int x = 80, y = 43;
    float z = (float)x/y;
    printf("%f\n", z);
}
```

De esta manera no se realizará el redondeo del resultado de la división.

3.4. Variables globales, constantes y estáticas

Una variable global es aquella que puede ser accedida en todo el programa, son declaradas fuera de las funciones. Ejemplo:

```
#include<stdio.h>
int x = 80;
int main() {
    printf("%d\n", x);
}
```

Una variable constante es una variable que no puede ser modificada, se declaran con la palabra reservada `const`:

```
#include<stdio.h>
int main() {
    const int x = 80;
    x = 99;
}
```

Este programa no puede compilarse ya que se intenta modificar la constante:

```
[chepe@chepeman ejemplo]$ gcc const.c -o const
const.c: In function ‘main’:
const.c:4:7: error: assignment of read-only variable ‘x’
     x = 99;
       ^
[chepe@chepeman ejemplo]$
```

Las variables estáticas se declaran con la palabra reservada `static`. Éstas tienen un periodo de vida que ocupa todo el tiempo de ejecución del programa. Para ilustrar su funcionamiento, se tiene el siguiente programa:

```
#include<stdio.h>
int * fun() {
    int x = 45;
    return &x;
}

int main() {
    printf("%d\n", *fun());
}
```

Se tiene la función `fun` que regresa un apuntador (referirse a la sección 9) a la variable local `x`. Al ejecutarse el programa se da un error indicando que no puede accederse a la variable local desde afuera de la función:

```
[chepe@chepeman ejemplo]$ ./static0
Segmentation fault (core dumped)
[chepe@chepeman ejemplo]$
```

Se modifica entonces la variable para que sea estática.

```
#include<stdio.h>
int * fun() {
    static int x = 45;
    return &x;
}

int main() {
    printf("%d\n", *fun());
}
```

De esta manera sí es posible acceder a ella fuera de la función:

```
[chepe@chepeman ejemplo]$ ./static1
45
[chepe@chepeman ejemplo]$
```

Para entender mejor el funcionamiento de estas variantes de variables, se tiene el programa siguiente:

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

```c
int a = 1;
const int b = 1;
static int c;
static int d = 1;
static const int e = 1;

int main() {
    int v = 1;
    const int w = 1;
    static int x;
    static int y = 1;
    static const int z = 1;
    printf("GLOBAL\n");
    printf("int: %p\n", &a);
    printf("const int: %p\n", &b);
    printf("static int: %p\n", &c);
    printf("static int inicializada: %p\n", &d);
    printf("static const int: %p\n", &e);

    printf("\nLOCAL\n");
    printf("int: %p\n", &v);
    printf("const int: %p\n", &w);
    printf("static int: %p\n", &x);
    printf("static int inicializada: %p\n", &y);
    printf("static const int: %p\n", &z);
}
```

Al correrlo con el depurador `gdb`, podemos observar las características de cada variante. Primero se usa el comando `(gdb) info files` para observar las distintas secciones del proceso:

```
0x0000000000400238 - 0x0000000000400254 is .interp
0x0000000000400254 - 0x0000000000400274 is .note.ABI-tag
0x0000000000400274 - 0x0000000000400298 is .note.gnu.build-id
0x0000000000400298 - 0x00000000004002b4 is .gnu.hash
0x00000000004002b8 - 0x0000000000400348 is .dynsym
0x0000000000400348 - 0x00000000004003a7 is .dynstr
0x00000000004003a8 - 0x00000000004003b4 is .gnu.version
0x00000000004003b8 - 0x00000000004003e8 is .gnu.version_r
0x00000000004003e8 - 0x0000000000400400 is .rela.dyn
0x0000000000400400 - 0x0000000000400460 is .rela.plt
0x0000000000400460 - 0x000000000040047a is .init
0x0000000000400480 - 0x00000000004004d0 is .plt
0x00000000004004d0 - 0x00000000004004d8 is .plt.got
0x00000000004004e0 - 0x0000000000400772 is .text
0x0000000000400774 - 0x000000000040077d is .fini
0x0000000000400780 - 0x00000000004007fc is .rodata
0x00000000004007fc - 0x0000000000400830 is .eh_frame_hdr
0x0000000000400830 - 0x0000000000400924 is .eh_frame
0x0000000000600e10 - 0x0000000000600e18 is .init_array
0x0000000000600e18 - 0x0000000000600e20 is .fini_array
0x0000000000600e20 - 0x0000000000600e28 is .jcr
0x0000000000600e28 - 0x0000000000600ff8 is .dynamic
0x0000000000600ff8 - 0x0000000000601000 is .got
0x0000000000601000 - 0x0000000000601038 is .got.plt
0x0000000000601038 - 0x0000000000601054 is .data
0x0000000000601054 - 0x0000000000601060 is .bss
```

El programa da como resultado la siguiente salida:

```
GLOBAL
int: 0x601048
const int: 0x400784
static int: 0x601058
static int inicializada: 0x60104c
static const int: 0x400788

LOCAL
int: 0x7fffffffd860
const int: 0x7fffffffd864
static int: 0x60105c
static int inicializada: 0x601050
static const int: 0x4007f8
```

Comparando los valores de las direcciones de memoria de las variables con el espacio de direcciones de cada sección, se puede observar que la variable `int` declarada globalmente se guarda en la sección `.data` del proceso, que es el mismo lugar donde se guardan las variables estáticas inicializadas declaradas global y localmente. Las variables constantes estáticas, así como la constante global, se encuentran en la sección `.rodata`. Las variables estáticas no inicializadas global y local se encuentran en la sección `.bss`. Finalmente, las variables locales `int` y `const int` tienen una dirección en memoria mucho mayor a las otras, lo cual indica que se guardan en la pila.

Se puede concluir de lo anterior que la sección `.data` almacena variables estáticas y globales que han sido inicializadas, la sección `.bss` contiene variables estáticas y globales no inicializadas, `.rodata` contiene constantes globales y estáticas (esta sección es de sólo lectura, o *read-only data*), y las variables no estáticas locales se guardan en la pila, más precisamente en el marco de pila de la función que las contiene. A diferencia de las constantes globales y estáticas, la constante local es guardada en un lugar que no es de sólo lectura, por lo que el mantenimiento de su integridad es verificado sólamente en tiempo de compilación.

Las variables almacenadas en las secciones `.data`, `.bss` y `.rodata`, al no estar en la pila, no dependen del contexto en el que se encuentren (los marcos de pila), es por esto que pueden ser accedidas desde cualquier función en cualquier momento.

4. Operadores

4.1. Aritméticos

- `X + Y`: Suma de dos valores.
- `X - Y`: Resta de dos valores.
- `X / Y`: División de dos valores.
- `X * Y`: Multiplicación de dos valores.
- `X % Y`: Operación de módulo (regresa residuo resultado de la división de dos números).
- `X++`: Operador unario. Incrementa en uno el valor de una variable.
- `X--`: Operador unario. Disminuye en uno el valor de una variable.

4.2. De comparación

- `X == Y`: Regresa 1 syss dos valores son iguales.
- `X != Y`: Regresa 1 syss dos valores no son iguales.
- `X > Y`: Regresa 1 syss si el primer operando es mayor que el segundo.
- `X < Y`: Regresa 1 syss si el primer operando es menor que el segundo.
- `X >= Y`: Regresa 1 syss si el primer operando es mayor o igual que el segundo.
- `X <= Y`: Regresa 1 syss si el primer operando es menor o igual que el segundo.

4.3. Lógicos

- `X && Y`: Operación lógica AND.
- `X || Y`: Operación lógica OR.
- `!X`: Negación.

4.4. Bitwise

- `X & Y`: Operación lógica AND a nivel de bits.
- `X | Y`: Operación lógica OR a nivel de bits.
- `~Y`: Negación a nivel de bits.
- `X ^ Y`: Operación lógica XOR a nivel de bits.
- `X << Y`: Corrimiento de bits hacia la izquierda.
- `X >> Y`: Corrimiento de bits hacia la derecha.

4.5. De asignación

- `X = Y`: A una variable se le asigna el valor resultante de una expresión. Regresa el valor asignado.
- `X ◇= Y`: Funciona como una asignación normal, pero el signo de igual es precedido por otro operador `◇:=[+-*/%»«&^|]`, donde la expresión `X ◇= Y` se interpreta como `X = X ◇ Y`.

4.6. Otros

- `sizeof(X)`: Regresa el tamaño de la variable X:
- `&X`: Operador de referencia. Regresa la dirección en memoria de la variable X.
- `*X`: Operador de indirección. Regresa el contenido de la variable a la que apunta el apuntador X. 9.

- `A ? B : C`: Regresa B si A es verdadera, de lo contrario regresa C.
- `A1, A2,..., An`: Operador n-ario que consiste en dos o más expresiones separadas por una coma, se evalúa cada una y al final se regresa el valor resultante de la última expresión.

5. Funciones

Una función tiene la siguiente estructura:

```
tipo nombre([parametros]) {

    cuerpo...
}
```

Donde:

```
parametros := (variable,)* variable
variable := tipo nombre
```

En la firma de la función, `tipo` es el tipo del valor que regresa la función, de ser `void` se indica que la función no regresa nada.

Los parámetros de la función son opcionales e indican el número de parámetros que recibe la función, así como el tipo de cada uno. Una función con la firma:

```
tipo nombre(void)
```

indica explícitamente que no recibe ningún parámetro.

El valor de la función es regresado dentro del cuerpo con la palabra reservada `return`.

Ejemplo:

```c
#include<stdio.h>

float funA(float a, float b) {
    puts("Entrando a funcion funA");
    return a + b - 1;
}

int main() {
    float a = 0.5, b = 0.9;
    printf("%f", funA(a, b));
}
```

6. Control de flujo

6.1. if

Forma de la expresión condicional `if`:

```
if(condicion) {
    ...
} else if(condicion) {
    ...
} else {
    ...
}
```

Como se había indicado en la sección de operadores 4.6, hay casos en los que esta expresión puede simplificarse con el operador ternario `A ? B : C`.

6.2. while

Forma de la expresión `while`:

```
while(condicion) {
    ...
}
```

Mientras se cumpla la condición se ejecuta el código dentro del ciclo `while`. También se tiene la siguiente expresión:

```
do {
    ...
} while(condicion);
```

Se ejecuta una vez el código dentro del ciclo `do-while`, y después se sigue ejecutando mientras la condición se cumpla.

6.3. for

Forma de un ciclo `for`:

```
for(expresion; expresion; expresion) {
    ...
}
```

`for` toma tres expresiones separadas por un punto y coma, normalmente la primer expresión es una variable o una declaración de variable, la segunda expresión es una condición que mientras se cumpla, indica que se ejecute el código dentro del ciclo, y la tercera una expresión que modifica la variable declarada, e.g.:

```
int j;
for(int i = 20, j = 0; i > j; i-=2, j+=2) {
    ...
}
```

6.4. switch

Forma de `switch`:

```
switch(expresion) {
    case valor:
        ...
        break;
    case valor2:
        ...
        break;
    ...
    default:
        ...
}
```

`switch` toma una expresión y compara su valor con los valores definidos con la palabra `case`, si encuentra un valor al que es igual, se ejecuta el código que le sigue al `case` con el que coincidió hasta encontrar un `break`. En caso de no coincidir con ningún valor definido, se ejecuta el código de `default`.

7. Preprocesador

El preprocesador actúa sobre el archivo de código fuente antes de que empiece la etapa de compilación, utiliza las llamadas "directivas" para modificar el código fuente, cada una de éstas empieza con el signo "#". Éstas son algunas de las directivas existentes:

7.1. #define

Esta directiva se utiliza para la definición de *macros*, los cuales son utilizados para hacer una sustitución textual (lo cual no es lo mismo que hacer una asignación de variable). Pueden ser de dos formas:

```
#define NOMBRE expresion
#define fun(...) expresion
```

La primer forma indica que las apariciones de NOMBRE en el código fuente sean remplazadas por la expresion, por ejemplo:

```
#define PI 3.14159
int main() {
    printf("Valor de PI: %f", PI);
}
```

La segunda forma permite definir el macro de una función, la cual será remplazada por la expresión, en la cual se pueden manipular los parámetros recibidos. Ejemplo:

```
#define area_tri(b, h) (((b) * (h)) / 2.0)
int main() {
    printf("Area de triangulo: %f", area_tri(1 + 3, 2 * 3 + 4));
}
```

En este ejemplo se remplaza `area_triangulo(1 + 3, 2 * 3 + 4)` por la expresion `(((1 + 3) * (2 * 3 + 4)) / 2.0)`.

7.2. #undef

Esta directiva se utiliza para indicar que se olvide alguna sustitución definida con `#define`. Ejemplo:

```
#define PI 3.14159
#undef PI
```

7.3. #include

Directiva que es remplazada por el contenido de un archivo, se puede usar de dos maneras:

```
/* Indica que se busque el archivo en directorios predefinidos
por el sistema. */
#include<archivo.h>
/* Indica que se busque el archivo en el directorio actual. */
#include"archivo.h"
```

La primera forma se utiliza para utilizar archivos de cabecera del sistema, la segunda para archivos de cabecera creados por el programador. Un archivo de cabecera contiene declaraciones de funciones y definiciones de macros que pueden ser compartidos por varios archivos, y tienen la extensión `.h`.

7.4. Condicionales

Existen directivas que pueden ser usadas para incluir texto en el código fuente si se cumple alguna condición, éstas son `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else` y `#endif`.

`#ifdef` verifica si un macro está definido, por el contrario, `#ifndef` verifica si un macro no está definido.

Ejemplo:

```
#define X 100

#ifdef X
#if X > 150
...
#elif X > 100
...
#else
...
#endif
#endif
```

8. Arreglos

Los arreglos consisten en una colección de variables que tienen las mismas características, las cuales son guardadas de manera consecutiva en memoria. Se declaran de la siguiente manera:

```
tipo nombre[N];
```

También pueden ser inicializados al ser declarados:

```
int numeros[] = {3, 1, 3, 3, 7};
```

La declaración de un arreglo indica que al ejecutarse el programa se haga un espacio en la memoria de al menos `N * sizeof(tipo)` bytes para almacenar sus elementos.

Para acceder al valor número N dentro de un arreglo, se utiliza el operador de indirección de arreglos `[]`:

```
arreglo[N];
```

Ejemplo:

```
#include<stdio.h>
int main() {
    float arr[] = {1.0, 3.4, 55.21, 98.0};
    for(int i = 0; i < 4; i++) {
        printf("%f ", arr[i]);
    }
}
```

9. Apuntadores

Los apuntadores consisten en una variable que tiene por valor una dirección de memoria, en la cual está almacenado el contenido de una variable. Un apuntador tiene un tipo, que es el tipo de la variable a la que apunta. Se declaran de la siguiente manera:

```
tipo * nombre;
```

Para crear un apuntador a una variable, se debe de extraer la dirección en memoria de esta última con el operador `&`:

```
int x = 5;
int * px = &x;
```

Para saber cuál es el valor de la variable `x` a partir del apuntador `px`, se debe de utilizar el operador de indirección `*`:

```
printf("Valor de x:%d\n", *px);
```

Este operador indica que se obtenga el dato de tipo `int` que está almacenado en la dirección de memoria que la variable `px` tiene como valor. Ejemplo:

```c
#include<stdio.h>
int main() {
    int x = 25;
    int y = x;
    int * px = &x;
    x = 10;
    printf("Valor de x: %d\n", x);
    printf("Valor de y: %d\n", y);
    printf("Valor de px: %p\n", px);
    printf("Valor de la variable a la que apunta px: %d\n", *px);
}
```

Para imprimir un apuntador se utiliza el especificador de formato "`%p`". El resultado de la ejecución de este programa es el siguiente:

```
[chepe@chepeman ejemplo]$ ./prueba2
Valor de x: 10
Valor de y: 25
Valor de px: 0x7ffc96853dd8
Valor de la variable a la que apunta px: 10
[chepe@chepeman ejemplo]$
```

Se puede observar que la variable `y` tiene el valor inicial de `x`, ya que en su declaración se le asignó una copia de éste, mientras que el apuntador `px`, ya que está apuntando al valor de `x`, al imprimirse con el operador de indirección muestra el valor actual de `x`. El valor que está almacenado en sí en `px`, como puede apreciarse, es un número largo que indica que se trata de una dirección en memoria.

Un apuntador de tipo `void` es utilizado para declarar apuntadores genéricos, es decir, puede ser usado como un apuntador de cualquier tipo. Ejemplo:

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

void * fun(int i) {
    if(i == 1)
        return malloc(sizeof(int));
    else
        return malloc(sizeof(char));
}

int main() {
    char * c = fun(2);
    *c = 'A';
    printf("Valor de c: %c\n", *c);
    int * d = fun(1);
    *d = 13;
    printf("Valor de d: %d\n", *d);
}
```

Resultado:

```
[chepe@chepeman ejemplo]$ ./prueba4
Valor de c: A
Valor de d: 13
[chepe@chepeman ejemplo]$
```

Cabe destacar que el tamaño de un apuntador depende de la arquitectura en la que se esté trabajando, será de 4 bytes en una arquitectura de 32 bits, y de 8 bytes en una de 64.

9.1. Relación entre arreglos y apuntadores

La variable que se utiliza para nombrar a un arreglo puede usarse de manera similar a un apuntador, y viceversa, por ejemplo, se puede utilizar un apuntador como se utiliza un arreglo para acceder a un elemento en una colección:

```c
#include<stdio.h>
int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int * p = arr;
    printf("%d\n", p[2]);
}
```

```
[chepe@chepeman ejemplo]$ ./apuntador
3
[chepe@chepeman ejemplo]$
```

Y la variable de un arreglo puede imprimirse como se imprime un apuntador, mostrando el lugar en memoria en donde están almacenados sus elementos:

```c
#include<stdio.h>
int main() {
    int arr[] = {1,2,3,4,5};
    printf("%p\n", arr);
}
```

```
[chepe@chepeman ejemplo]$ ./arreglo
0x7fff086859d0
[chepe@chepeman ejemplo]$
```

Lo anterior se debe a que la variable de un arreglo puede ser "convertida" a un apuntador, por eso puede ser impresa como tal. A demás, el operador de indirección de arreglos `a[N]` está definido en término de apuntadores, por lo que es equivalente a `*(a + N)`, razón por la cual también puede ser aplicado a los apuntadores.

A pesar de lo anterior, los arreglos y los apuntadores son esencialmente distintos, ya que una vez declarado un arreglo, no se puede modificar el lugar en memoria en el que está almacenado, mientras que el valor de un apuntador puede modificarse libremente. Por ejemplo, la compilación de este programa dará un error, ya que no es posible modificar la variable de un arreglo:

```c
int main() {
    int c[] = {1,2,3,4,5};
    int x = 9;
    c = &x;
}
```

```
[chepe@chepeman ejemplo]$ gcc arreglo.c -o arreglo
arreglo.c: In function 'main':
arreglo.c:4:7: error: assignment to expression with array type
     c = &x;
       ^
[chepe@chepeman ejemplo]$
```

9.2. Aritmética de apuntadores

Al incrementar o disminuir el valor de un apuntador en uno (`p++` o `p--`), no siempre se obtiene el resultado intuitivo de modificar en uno el valor del apuntador, como sucede con otros tipos, más bien se incrementa o disminuye en N valores, dependiendo del tamaño del tipo del apuntador. Esto es lo que permite utilizar el operador de indirección de un arreglo, es decir, `a[i]`, que como se indicó en la sección anterior, es equivalente a `*(a + i)`. Si el valor del apuntador (que es una dirección de memoria) aumentara en uno, entonces se tendría problemas para acceder a elementos de tipos con tamaño mayor a uno.

Por ejemplo, suponiendo que ese tiene el arreglo `int a[] = {1, 2, 3}`, para acceder al primer elemento se utiliza `a[0]`, que es equivalente a `*a`, pero si queremos acceder al siguiente elemento, entonces el valor de `*(a + 1) = a[1]` debe de ser igual a `a + 4`, ya que cada entero consta de 4 bytes, así que con cada incremento del apuntador, se debe de incrementar el valor de la dirección de memoria en 4, si no se estaría leyendo el segundo byte del primer elemento. Por ejemplo, se tiene el siguiente programa:

```c
#include<stdio.h>

int main() {
    int iarr[] = {1, 2, 3, 4};
    char carr[] = {'a', 'b', 'c', 'd'};
    long long int llarr[] = {50, 51, 52, 54};
    for(int i = 0; i < 4; i++) {
        printf("Valor de apuntador: %p, valor al que apunta: %d\n",
               iarr+i, *(iarr+i));
        printf("Valor de apuntador: %p, valor al que apunta: %c\n",
               carr+i, *(carr+i));
        printf("Valor de apuntador: %p, valor al que apunta: %lld",
               llarr+i, *(llarr+i));
        puts("\n");
    }
}
```

Este programa itera tres arreglos de distintos tipos e imprime el valor de los apuntadores en cada iteración, así como el valor al que apuntan. Ejecución del programa:

```
[chepe@chepeman ejemplo]$ ./apuntadores
Valor de apuntador: 0x7ffdb5a00040, valor al que apunta: 1
Valor de apuntador: 0x7ffdb5a00070, valor al que apunta: a
Valor de apuntador: 0x7ffdb5a00050, valor al que apunta: 50

Valor de apuntador: 0x7ffdb5a00044, valor al que apunta: 2
Valor de apuntador: 0x7ffdb5a00071, valor al que apunta: b
Valor de apuntador: 0x7ffdb5a00058, valor al que apunta: 51

Valor de apuntador: 0x7ffdb5a00048, valor al que apunta: 3
Valor de apuntador: 0x7ffdb5a00072, valor al que apunta: c
Valor de apuntador: 0x7ffdb5a00060, valor al que apunta: 52

Valor de apuntador: 0x7ffdb5a0004c, valor al que apunta: 4
Valor de apuntador: 0x7ffdb5a00073, valor al que apunta: d
Valor de apuntador: 0x7ffdb5a00068, valor al que apunta: 54

[chepe@chepeman ejemplo]$
```

Se puede observar que en cada iteración el apuntador de tipo `int` se incrementa en 4, el de tipo `char` en 1 y el de tipo `long long int` en 8. [3]

10. Cadenas

En C no existe el tipo `string` de la misma manera que existe en otros lenguajes, las cadenas se manejan como arreglos de caracteres, los cuales se imprimen usando el especificador de formato "`%s`", que indica que se impriman los datos almacenados en el arreglo hasta que se encuentre un terminador de cadena (un 0).

Una cadena puede inicializarse de las siguientes maneras:

```
char cadena[] = "hola";
char * cadena = "hola";
```

La primera forma crea un arreglo del tamaño de la cadena de manera local, la segunda crea un apuntador la cadena. Se puede apreciar la diferencia entre ambas formas de incialización con el siguiente ejemplo:

```
#include<stdio.h>
int main() {
    char * cadena1 = "hola1";
    printf("Valor de cadena1: %p\n", cadena1);
    char cadena2[] = "hola2";
    printf("Valor de cadena2: %p\n", cadena2);
}
```

Resultado de ejecución:

[3] Se recomienda leer la parte de arreglos y apuntadores del libro *Hacking: The Art of Exploitation* de Jon Erickson, donde se explican estos temas a profundidad.

```
[chepe@chepeman ejemplo]$ ./prueba2
Valor de cadena1: 0x40072c
Valor de cadena2: 0x7ffff4666570
[chepe@chepeman ejemplo]$
```

Se puede observar que `cadena1` tiene un valor mucho menor a `cadena2`, ésto se debe a que la manera en la que se declaró `cadena1` indica que se trata de una cadena estática constante (es decir, sus datos no pueden ser modificados), por lo que su contenido es guardado en la sección `.rodata` del archivo binario, mientras que el contenido de `cadena2` se guardan en la sección `.text`, y durante la ejecucion del programa, se almacenará en la pila, cuyo espacio de direcciones en el proceso tiene valores mayores a los de `.rodata`.

11. Asignación dinámica de memoria

Durante la ejecución de un programa, los datos de un arreglo declarado dentro de una función son almacenados dentro del marco de pila de la función, el cual se vuelve inválido una vez ésta termina de ejecutarse. Por esta razón, en C no es válido que una función regrese un arreglo declarado localmente, ya que al salir de la función, el marco de pila que previamente le correspondía a la función será modificado por el marco de pila de otra función que se ejecute posteriormente, dañando la integridad de los datos del arreglo. Ejemplo:

```c
#include<stdio.h>
char * fun() {
    char cadena[] = "ejemplo";
    return cadena;
}
int main() {
    puts(fun());
}
```

```
[chepe@chepeman ejemplo]$ ./segmentation
Segmentation fault (core dumped)
[chepe@chepeman ejemplo]$
```

Se tiene como resultado de la ejecución del programa el mensaje de error `segmentation fault` [4].

Este problema puede evitarse haciendo uso de la asignación dinámica de memoria, que en vez de guardar los arreglos como variables locales, los almacena en el *heap* del proceso, lugar en donde los datos no son modificados a menos que sea indicado explícitamente por las instrucciones.

La asignación dinámica de memoria se realiza por medio de las funciones `malloc`, `realloc`, `calloc` y `free`, las cuales están definidas en `stdlib.h`.

[4] El mensaje de error `segmentation fault` indica que se trató de acceder o modificar una dirección de memoria inválida. En el caso de este programa, algunos compiladores no hacen que se regrese un `segmentation fault`, en el ejemplo mostrado el compilador hace que `fun` regrese un 0 (una dirección de memoria inválida), pero otros compiladores sí permiten que `fun` regrese el apuntador a la cadena, por lo que si bien se puede acceder a los datos del arreglo local, éstos tendrán valores indefinidos ya que su integridad no puede ser asegurada.

11.1. malloc

Esta función recibe un argumento, el cual indica el número de bytes a ser asignados en un espacio contiguo de memoria en el heap, se regresará un apuntador genérico al principio de tal espacio, este apuntador debe de convertirse al tipo deseado. Ejemplo:

```
int * p = (int*) malloc(15 * sizeof(int));
```

Se indica que quieren asignarse 15 datos de tipo `int` de manera contigua, cuyo tamaño total será la multiplicación de 15 por el tamaño de cada entero (en total 60 bytes).

11.2. calloc

Funciona de manera similar a `malloc`, pero recibe dos argumentos, el primero es el número de bloques contiguos a ser asignados, el segundo es el tamaño de cada bloque. A demás, el espacio que asigna en el heap es inicializado en ceros. Ejemplo:

```
int * p = (int*) calloc(15, sizeof(int));
```

11.3. free

Libera el espacio de memoria asignado en el heap con las funciones anteriores para que pueda ser reutilizada. Recibe un apuntador al espacio de memoria. Esta función permite que no se gaste memoria de más, así que es conveniente usarla una vez se deje de utilizar un espacio de memoria asignado dinámicamente. Ejemplo:

```
int * p = (int*) calloc(15, sizeof(int));
free(p);
```

11.4. realloc

Cambia el tamaño del espacio de memoria asignada con las funciones anteriores. Si se da un fallo en la asignación de memoria, se regresa 0, de lo contrario se regresa un apuntador al inicio del espacio. Ejemplo:

```
int * p = (int*) calloc(15, sizeof(int));
p = realloc(p, 30 * sizeof(int));
```

Escribir lo siguiente:

```
p = realloc(p, 0);
```

es equivalente a

```
free(p);
```

Así mismo, escribir

```
int * p = (int *) realloc(0, 15 * sizeof(int));
```

es igual a

```
int * p = (int *) malloc(15 * sizeof(int));
```

12. Estructuras

Una estructura es una colección de datos que pueden ser de distinto tipo. Se definen de la siguiente manera:

```
struct identificador {
  char c;
  int i;
  int * p;
};
```

De esta manera puede declararse una variable de tipo `struct identificador`:

```
struct identificador id;
```

Para acceder a las variables de `id` se usa un punto, de esta manera podemos también asignarles valores:

```
id.c = 'A';
id.i = 0x777;
id.p = &id.i;
```

Es posible también declarar apuntadores de tipo `struct identificador`:

```
struct identificador * idp = malloc(sizeof(struct identificador));
```

En este caso, para acceder a las variables de `idp`, se utiliza la siguiente notación:

```
idp->c = 'B';
idp->i = 999;
idp->p = &(idp->i);
```

También es posible utilizar `typedef` para simplificar la declaración de variables de estructuras:

```
typedef struct identificador {
  char c;
  int i;
  int * p;
} identificador;

identificador id;
id.c = 'O';
```

Es posible inicializar una estructura de las siguientes maneras:

```
identificador id1 = {'r', 2, 0};
identificador id2 = {c:'r', i:2};
```

Una de las utilidades de las estructuras es la posiblidad de crear estructuras de datos como árboles, listas, etc.

13. Paso de argumentos

Para leer argumentos que sean pasados al programa desde su ejecución en terminal, se le agregan unos parámetros extra a la función `main`, los cuales consisten en el número de argumentos recibidos,

así como el arreglo que contiene a los argumentos. El primer miembro del arreglo de argumentos siempre es el nombre del ejecutable. Ejemplo:

```
#include<stdio.h>
int main(int argc, char * argv[]) {
    for(int i = 0; i < argc; i++)
        printf("%s\n", argv[i]);
}
```

Ejecución:

```
[chepe@chepeman ejemplo]$ ./prueba3 uno dos tres cuatro
./prueba3
uno
dos
tres
cuatro
[chepe@chepeman ejemplo]$
```

14. Manejo de archivos

Los archivos pueden manejarse por medio de las llamadas al sistema `open`, `write`, `read` y `close`.

- `open`: Abre un archivo y regresa su descriptor de archivo. Recibe el nombre de éste junto con unas banderas que indican el modo de acceso en el que se quiere abrir:
  - `O_RDONLY`: Modo de sólo lectura.
  - `O_WRONLY`: Modo de sólo escritura, se sobreescribe el archivo.
  - `O_RDWR`: Modo de lectura y escritura.
  - `O_CREAT`: Se crea el archivo si no existe.
  - `O_EXCL`: Evita que se cree el archivo si ya existe.
  - `O_APPEND`: En el modo de escritura, se agrega el contenido al final del archivo.

  También recibe opcionalmente el modo de permisos del archivo.
- `read`: Lee n bytes del archivo y los copia en en un buffer. Regresa 0 si se alcanzó el final del archivo, -1 cuando hay un error o el número de bytes que fueron leídos en otro caso.
- `write`: Copia n bytes desde un buffer hacia un archivo. Regresa el número de byes copiados, o -1 en caso de error
- `close`: Cierra el archivo, recibe el descriptor de archivo. Regresa 0 si no hubo error y -1 de lo contrario.

Ejemplo:

```
#include<fcntl.h>
#include<unistd.h>
#include<stdio.h>
#include<string.h>
```

```c
int main(int argc, char * argv[]) {
    if(argc < 2) {
        printf("No se indico archivo\n");
        return 1;
    }
    int fd = open(argv[1], O_WRONLY | O_APPEND | O_CREAT, 0644);
    write(fd, "HOLA!!\n", 7);
    close(fd);
    char buf[10];
    memset(buf, 0, 10);
    int fdR = open(argv[1], O_RDONLY);
    while(read(fdR, buf, 9)) {
        printf("%s", buf);
        memset(buf, 0, 10);
    }
    close(fdR);
}
```

Resultado:

```
[chepe@chepeman ejemplo]$ ./prueba3 archivo
HOLA!!
[chepe@chepeman ejemplo]$ ./prueba3 archivo
HOLA!!
HOLA!!
[chepe@chepeman ejemplo]$ ./prueba3 archivo
HOLA!!
HOLA!!
HOLA!!
[chepe@chepeman ejemplo]$
```

15. Bibliotecas principales

Las bibliotecas de C pueden ser importadas y utlizadas por los programadores. Para poder utilizar una biblioteca, debe de incluirse su archivo de encabezados con la directiva `#include`. Algunos ejemplos de bibliotecas importantes son los siguientes:

15.1. stdio.h

Contiene declaraciones de funciones para manejo de entrada y salida. Estos son algunos de los macros y tipos que define:

- `FILE`: Tipo de un objeto que contiene información sobre el control de un flujo de datos (*stream*). Permite manejar archivos utilizando funciones de más alto nivel que si se utilizaran las llamadas al sistema descritas en 14.
- `EOF`: Constante entera negativa que indica que se ha alcanzado el final de un archivo en su lectura.
- `FILENAME_MAX`: La longitud máxima que puede tener el nombre de un archivo.

- `FOPEN_MAX`: El número máximo de archivos que tienen la garantía de poder ser abiertos concurrentemente.
- `SEEK_CUR, SEEK_END, SEEK_SET`: Constantes enteras que se utilizan para controlar las acciones de la función `lseek`.
- `stdin, stdout, stderr`: Objetos predefinidos de tipo `FILE*`, que se refieren respectivamente a la entrada estándar, la salida estándar y la salida de error.

Éstas son algunas funciones definidas:

`int remove(const char *nombre_archivo)`
Elimina el archivo cuyo nombre recibe como parámetro.

`int rename(const char *viejo, const char *nuevo)`
Cambia el nombre del archivo con nombre `viejo` por el nombre `nuevo`.

`FILE *fopen(const char *archivo, const char *modo)`
Esta función es de más alto nivel que `open`, abre el archivo indicado, con el modo requerido, y regresa un apuntador de tipo `FILE*`. Se pueden utilizar los siguientes modos:

- `"r"`: Lectura de archivo de texto.
- `"rb"`: Lectura de archivo binario.
- `"r+"`: Lectura y escritura de archivo de texto.
- `"r+b"/"rb+"`: Lectura y escritura de archivo binario.
- `"w"`: Escritura de archivo de texto. Sobreescribe archivo.
- `"wb"`: Escritura de archivo binario. Sobreescribe archivo.
- `"w+"`: Lectura y escritura de archivo de texto. Sobreescribe archivo.
- `"w+b"/"wb+"`: Lectura y escritura de archivo binario. Sobreescribe archivo.
- `"a"`: Escritura al final de archivo de texto.
- `"ab"`: Escritura al final de archivo binario.
- `"a+"`: Lectura y escritura al final de de archivo de texto.

`FILE *tmpfile(void)`
Crea un archivo temporal y regresa un apuntador a su flujo de datos de tipo `FILE*`. Al cerrarse el flujo de datos, se elimina el archivo.

`int fclose(FILE *stream)`
Cierra un archivo abierto. Regresa 0 si no hay errores, -1 de lo contrario.

`int printf(const char *formato, ...)`
Imprime a `stdout` la cadena con el formato especificado, modificando en éste los especificadores

de formato por un número no definido de argumentos.

`int fprintf(FILE *archivo, const char *formato, ...)`
Funciona como `printf`, pero escribe la cadena de salida en el archivo especificado.

`int sprintf(char *s, const char *formato, ...)`
Funciona como `printf`, pero escribe la cadena de salida en el arreglo de caracteres específicado.

`int fscanf(FILE *archivo, const char *formato, ...)`
`int sscanf(const char *s, const char *formato, ...)`
`int scanf(const char *formato, ...)`
Estas funciones leen un flujo de datos (`fscanf`), una cadena (`sscanf`) o `stdin` (`scanf`), y guardan los caracteres que coincidan con un formato, con el tipo de dato definido por los especificadores de formato, en los últimos apuntadores que son pasados como argumentos.

`size_t fwrite(const void * ptr, size_t tam, size_t n, FILE * archivo)`
Escribe `n` elementos de tamaño `tam`, copiados desde `ptr` hacia `archivo`.

`size_t fread(void * ptr, size_t tam, size_t n, FILE * arhcivo)`
Lee `n` elementos de tamaño `tam` desde `archivo` y los copia en `ptr`.

Ejemplo:

```c
#include<stdio.h>

int main(int argc, char * argv[]) {
    if(argc < 2) {
        fprintf(stderr, "No se indico archivo\n");
        return 1;
    }
    char * usuarios[] = {"juan", "jorge", "jose"};
    int contrasenas[] = {31337, 999, 12345};
    FILE * fp = fopen(argv[1], "w");
    for(int i = 0; i < 3; i++)
        fprintf(fp, "%s %d\n", usuarios[i], contrasenas[i]);
    fclose(fp);

    fp = fopen(argv[1], "r");
    char usuario[10];
    int contrasena;
    while(fscanf(fp, "%s %d", usuario, &contrasena) != EOF)
        printf("El usuario %s tiene la contrasena %d\n",
               usuario, contrasena);
}
```

Ejecución:

```
[chepe@chepeman ejemplo]$ ./stdio archivo
El usuario juan tiene la contraseña 31337
El usuario jorge tiene la contraseña 999
El usuario jose tiene la contraseña 12345
[chepe@chepeman ejemplo]$
```

15.2. string.h

Define funciones que permiten manejar cadenas. Algunas de estas funciones son:

`void * memcpy(void *s1, const void *s2, size_t n)`
Copia `n` bytes desde `s2` a `s1`.

`char * strcpy(char *s1, const char *s2)`
`char * strncpy(char *s1, const char *s2, size_t n)`
Copian la cadena `s2` a la cadena `s1`, incluyedo el terminador de cadena. `strncpy` copia a lo más `n` bytes, si `s2` tiene menos de `n` caracteres, entonces rellena el resto con ceros.

`char * strcat(char *s1, const char *s2)`
`char * strncat(char *s1, const char *s2, size_t n)`
Concatenan `s1` y `s1` en `s1`, siempre se escribe al final de la concatenación un terminador de cadena. `strncat` copia a lo más `n` cracteres de `s2` a `s1`.

`int strcmp(const char *s1, const char *s2)`
`int strncmp(const char *s1, const char *s2, size_t n)`
Comparan dos cadenas, `strncmp` compara a lo más `n` caracteres. Regresan cero si las cadenas son iguales, de lo contrario, regresan un entero menor a cero si `s1` es lexicográficamente menor a `s2`, o un entero mayor a cero si `s1` es lexicográficamente mayor a `s2`.

`void * memset(void *s, int c, size_t n)`
Le asigna a los primeros `n` bytes de `s` el valor `c`.

`size_t strlen(const char *s)`
Regresa la longitud de `s` (cuenta el número de caracteres antes de encontrar un terminador de cadena).

15.3. arpa/inet.h

Este archivo consta de definiciones para realizar operaciones de internet, como la manipulación se sockets.

Se define una estructura que se utiliza para identificar a un host, consiste de una familia de direcciones (siempre se usa AF_INET), una estructura que contiene una dirección IPv4 y un puerto:

```
struct sockaddr_in {
    sa_family_t    familia;
    in_port_t      puerto;
```

```
    struct in_addr direccion;
};
```

int socket(int dominio, int tipo, int protocolo)
Crea un socket y regresa su descriptor de archivo. El `dominio` especifica un dominio de comunicación (la familia de protocolos que serán utilizados para entablar la comunicación). El `tipo` especifica el tipo de socket a ser usado (orientado o no a la conexión, transmisión en crudo, etc.). El `protocolo` especifica el protocolo que se usará para el socket, normalmete existe un protocolo por familia, por lo que basta con utilizar el valor 0.

int setsockopt(int sockfd, int nivel, int opnombre, const void *opval, socklen_t optam)
Asigna una opción a un socket. Recibe el descriptor de archivo del socket, un nivel que indica a qué nivel recide la opción (por ejemplo, a nivel de socket se utiliza `SOL_SOCKET`). `opnombre` es el nombre de la opción, `opval` el valor asignado a la opción y `optam` el tamaño de la opción. Ejemplos de opciones:

- `SO_REUSEADDR`: Permite que se vuelva a usar el socket.
- `SO_REUSEPORT`: Permite utilizar direcciones y puertos duplicados.
- `SO_DEBUG`: Permite registrar información de depuración.
- `SO_BROADCAST`: Permite mandar mensajes de tipo *broadcast*.

uint16_t htons(uint16_t puerto)
Convierte `puerto` de LSB a MSB.

uint16_t ntohs(uint16_t netshort)
Convierte `puerto` de MSB a LSB.

char *inet_ntoa(struct in_addr in)
convierte la dirección `in` a una cadena de la dirección IPv4 en notación decimal punteada.

int inet_pton(int af, const char *src, void *dst)
Convierte una cadena a una estructura de dirección IP. El primer argumento se refiere al protocolo, ya sea IPv4 (AF_INET) o IPv6 (AF_INET6). `src` es la cadena a convertir, `dst` es la estructura en la cual se guardarán los valores, para IPv4 es de tipo `struct in_addr`, para IPv6 es `struct in6_addr`.

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlon)
Asigna la dirección IP y puerto especificados por `addr` al socket con el descriptor `sockfd`. `addrlon` es el tamaño de `addr`.

int listen(int sockfd, int backlog)
Indica que el socket con el descriptor `sockfd` se ponga a la escucha de conexiones. El argumento `backlog` especifica el tamaño máximo posible de la cola de conexiones en espera por `sockfd`.

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlon)
Extrae la primer conexión en la cola de conexiones pendientes del socket en espera con el descriptor

`sockfd`, la estructura a la que apunta `addr` será rellenada con los datos del cliente, y `addrlon` con el tamaño de `addr`. Regresa el descriptor de archivo del socket aceptado, -1 si se da un error.

`int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen)`
Conecta el socket con el descriptor `sockfd` al host especificado por `addr`. Regresa -1 en caso de error, de lo contrario regresa 0.

`ssize_t recv(int sockfd, void *buf, size_t n, int banderas)`
Recibe un mensaje de `n` bytes desde el socket con el descriptor `sockfd` y lo copia en `buf`. Regresa el número de bytes que recibió desde el socket, -1 si hubo un error o 0 cuando se ha terminado la conexión.

`ssize_t send(int sockfd, const void *buf, size_t n, int banderas)`
Envía `n` bytes del mensaje `buf` a través del socket con el descriptor `sockfd`. Regresa el número de bytes enviados, o -1 en caso de error.

En el siguiente ejemplo se creó un servidor que acepta conexiones de clientes, uno a la vez, y recibe mensajes que imprime en la salida estándar:

`servidor.c`

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<unistd.h>

void error(char * mensaje) {
    fprintf(stderr, "%s\n", mensaje);
    exit(1);
}

int main(int argc, char * argv[]) {
    if(argc < 2) {
        fprintf(stderr, "Uso: %s <puerto>\n", argv[0]);
        return 1;
    }
    int sockfd, cliente_sockfd, opt = 1;
    struct sockaddr_in host_addr, cliente_addr;
    socklen_t sin_tam = sizeof(struct sockaddr_in);
    char msg[256];
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        error("Error al crear el socket");
    if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &opt,
        sizeof(int)) < 0)
        error("Error en setsockopt\n");
```

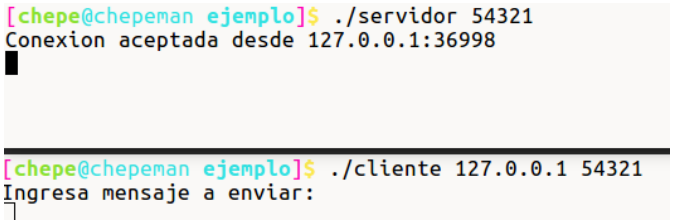
```
    host_addr.sin_family = AF_INET;
    host_addr.sin_port = htons(atoi(argv[1]));
    host_addr.sin_addr.s_addr = INADDR_ANY;
    if (bind(sockfd, (struct sockaddr *)&host_addr,
        sizeof(struct sockaddr)) < 0)
        error("Error en bind\n");
    if (listen(sockfd, 5) < 0)
        error("Error en listen\n");
    while(1) {
        cliente_sockfd = accept(sockfd,
                                (struct sockaddr *)&cliente_addr,
                                &sin_tam);
        if(cliente_sockfd < 0)
            error("Error en accept\n");
        printf("Conexion aceptada desde %s:%d\n",
            inet_ntoa(cliente_addr.sin_addr),
                      ntohs(cliente_addr.sin_port));
        int bytes_recibidos = recv(cliente_sockfd, msg, 256, 0);
        printf("Tamano de mensaje: %d\n", bytes_recibidos);
        printf("Mensaje: %s\n\n", msg);
        close(cliente_sockfd);
    }
    return 0;
}
```

cliente.c

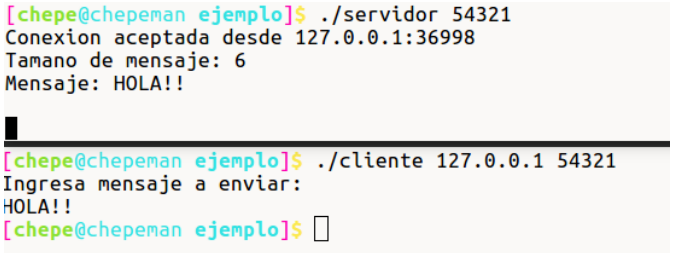
```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<unistd.h>

void error(char * mensaje) {
    fprintf(stderr, "%s\n", mensaje);
    exit(1);
}

int main(int argc, char * argv[]) {
    if(argc < 3) {
        fprintf(stderr, "Uso: %s <direccion IP> <puerto>\n",
                argv[0]);
        return 1;
    }
    int sockfd;
    struct sockaddr_in host_addr;
```

```
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        error("Error al crear socket\n");
    memset(&host_addr, 0, sizeof(host_addr));
    host_addr.sin_family = AF_INET;
    host_addr.sin_port = htons(atoi(argv[2]));
    if(inet_pton(AF_INET, argv[1], &host_addr.sin_addr) <= 0)
        error("Error al leer direccion IP de servidor\n");
    if (connect(sockfd, (struct sockaddr *)&host_addr,
        sizeof(host_addr)) < 0)
        error("Error al conectarse a servidor\n");
    char msg[256];
    memset(msg, 0, 256);
    printf("Ingresa mensaje a enviar:\n");
    scanf("%s", msg);
    send(sockfd , msg, strlen(msg) , 0);
    return 0;
}
```

Se ejecutó el programa del servidor seguido del cliente, el servidor escucha conexiones a través del puerto 54321, y acepta la conexión del cliente, el cual espera la lectura de un mensaje por la entrada estándar.



Una vez ingresado un mensaje del lado del cliente, éste es enviado al servidor, el cual lo imprime al recibirlo.



Referencias

[1] Kernighan, B. & Ritchie, D. (1988). *The C Programming Language* (2a ed). Estados Unidos: Prentice-Hall.

[2] Banahan, M. & Brady, D. & Doran, M. (1991). *The C Book* (2a ed). Estados Unidos: Addison Wesley.

[3] Erickson, J. (2008). *Hacking: The Art of Exploitation* (2a ed). Estados Unidos: No Starch Press.

[4] Stallman, R. & McGrath, R. (Abr 2000) *An Introduction to Makefiles*. Sitio web: `ftp://ftp.gnu.org/old-gnu/Manuals/make-3.79.1/html_chapter/make_2.html`