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 550 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.

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 último estándar ISO, publicado en 1999.

Si bien a esta altura la mayoría de los compiladores de C soportan prácticamente el estándar completo de C99 [3], 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/n1256.pdf
[2] http://www.ioccc.org/
[3] 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, por omisión **`unsigned`**, de tamaño de 1 byte.

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

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. 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 [4]:

```
puntero_estructura->nombre
```

5.4. 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;
```

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

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.

5.5. 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.

5.6. 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.
```

5.7. 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[5]. 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. usar, 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.

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

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 [6] [7].

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 [8] 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
```

[6] http://linux.die.net/man/
[7] http://www.linuxinfor.com/spanish/man3/index.html
[8] Codeblocks, Geany, Anjuta, Kdevelop, etc

```
$ # 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.

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.