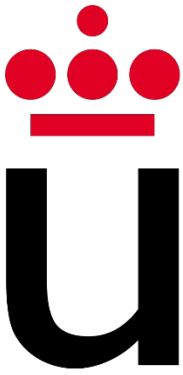


Universidad
Rey Juan Carlos

Apuntes de la Asignatura

Introducción a la Programación

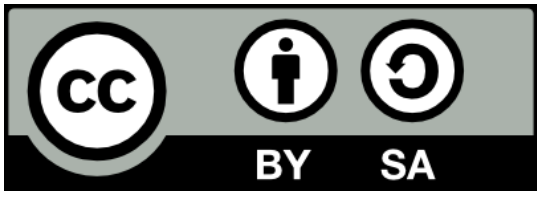
Curso 2024/2025

Grados de impartición:

- Grado en Ingeniería de Computadores
- Grado en Ingeniería de la Ciberseguridad

Autores:

Diego Hortelano, Gerardo Reyes, Manuel Rubio



©2024 Diego Hortelano Haro, Gerardo Reyes Salgado, Manuel Rubio Sánchez. Algunos derechos reservados. Este documento se distribuye bajo la licencia "Atribución/Reconocimiento-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en https://creativecommons.org/licenses/by-sa/4.0/deed.es.

Índice de contenidos

©2024 Diego Hortelano Haro, Gerardo Reyes Salgado, Manuel Rubio Sánchez. Algunos derechos reservados. Este documento se distribuye bajo la licencia "Atribución/Reconocimiento-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en https://creativecommons.org/licenses/by-sa/4.0/deed.es.

Tema 1: Introducción a la Programación

En este tema se presentan una serie de conceptos básicos relacionados con la programación y la resolución de problemas que nos aportarán las bases para los siguientes temas.

1. Resolución de problemas

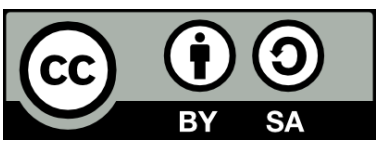
Aprender a resolver problemas de manera sistemática y lógica es un aspecto fundamental a la hora de programar. La programación no solo trata de escribir código, antes de ello es necesario entender el problema y diseñar una solución eficiente. Los pasos clave para la resolución de problemas en programación son:

1. Entender el problema. El primer paso antes de intentar proponer cualquier solución es comprender completamente el problema. Esto incluye identificar el resultado esperado, los datos de entrada disponibles y su formato y qué restricciones tiene el problema.
2. Dividir el problema. Si nos enfrentamos a un problema complejo puede resultar de gran utilidad dividirlo en varios subproblemas que puedan ser abordados individualmente de manera más sencilla.
3. Diseñar un algoritmo. Un algoritmo es una secuencia de pasos que permite resolver un problema. Este algoritmo puede estar escrito en pseudocódigo y permite representar y visualizar la solución y detectar posibles errores de nuestra propuesta. En los apartados 2. Algoritmos y 7. Pseudocódigo se detallan estos conceptos.
4. Implementar la solución. Una vez diseñada la solución, se debe implementar la solución en el lenguaje de programación elegido, siempre siguiendo buenas prácticas de programación.
5. Depurar. Una vez implementado nuestro programa, es necesario probarlo utilizando diferentes datos de entrada, prestando especial atención a casos excepcionales (por ejemplo, si nuestro programa realiza una división entre dos números pedidos al usuario, debemos comprobar que controla correctamente si el usuario incluye un "0" en el divisor).
6. Optimizar. Después de comprobar que nuestro programa funciona correctamente, es posible revisarlo para tratar de optimizar su eficiencia y refinar nuestro código, haciéndolo más limpio y entendible.

2. Algoritmos

Un algoritmo es un conjunto ordenado de instrucciones que permiten hallar la solución de un determinado problema o realizar una determinada tarea. Para ello, el conjunto de instrucciones debe ser finito (para permitirnos alcanzar la solución del problema) y estar definido claramente (a mayor nivel de detalle más sencillo resulta seguir las instrucciones).

Esto hace que, dado un algoritmo, no sea necesario entender sus principios para poder ejecutarlo, es suficiente con ejecutar sus instrucciones en el orden indicado para resolver el problema.



©2024 Diego Hortelano Haro, Gerardo Reyes Salgado, Manuel Rubio Sánchez. Algunos derechos reservados. Este documento se distribuye bajo la licencia "Atribución/Reconocimiento-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en https://creativecommons.org/licenses/by-sa/4.0/deed.es.

2.1. Características de un algoritmo

Un algoritmo debe cumplir con las siguientes características:

1. **Finitud**: debe tener un número finito de instrucciones, así como un final bien definido.
2. **Claridad y precisión**: cada instrucción debe ser clara y precisa, no pudiendo existir ambigüedad en las instrucciones.
3. **Orden secuencial**: las instrucciones deben estar organizadas de manera lógica y secuencial, permitiendo ejecutarlas claramente desde el inicio hasta el final para resolver el problema.
4. **Entrada y salida**: un algoritmo puede tener cero o más entradas (datos iniciales necesarios para realizar las operaciones) y una o más salidas (resultados producidos por el algoritmo).
5. **Efectividad**: las instrucciones deben ser comprensibles y ejecutables, ya sea por una persona o por un computador.
6. **Determinismo**: dado una entrada determinada, un algoritmo siempre debe producir el mismo resultado.

2.2. Ejemplos de algoritmos

Como ejemplos de algoritmos se presentan tres: el algoritmo para extraer el valor mínimo de una lista, el algoritmo de Búsqueda Lineal y el algoritmo de Ordenamiento Burbuja (o Bubble Sort).

2.2.1. Algoritmo para extraer el valor mínimo

Este algoritmo retorna el elemento más pequeño de una lista de números.

Entrada: lista de n números naturales.

Salida: valor mínimo de la lista.

Pasos:

1. Toma el primer elemento de la lista y considéralo el mínimo.
2. Para cada uno de los $n - 1$ elementos restantes:
    a. Compáralo con el mínimo actual.
    b. Si el elemento actual es menor que el mínimo almacenado, actualiza el mínimo con el nuevo valor.

A continuación, se muestra un ejemplo paso a paso de este algoritmo, para la lista de entrada [7, 2, 4, 1, 9]:

1. Inicialmente, consideramos el primer elemento como el mínimo: $min = 7$.
2. Comparamos el mínimo con el segundo elemento de la lista, un 2:
    - Como $7 > 2$, actualizamos el mínimo: $min = 2$.
3. Comparamos el mínimo con el tercer elemento de la lista, un 4:
    - Como $2 < 4$, mantenemos el mínimo actual.
4. Comparamos el mínimo con el siguiente elemento de la lista, un 1:
    - Como $2 > 1$, actualizamos el mínimo: $min = 1$.
5. Comparamos el mínimo con el último elemento de la lista, un 9:

- Como $1 < 9$, mantenemos el mínimo actual.

6. Después de revisar toda la lista, el mínimo es 1.

2.2.2. Algoritmo de Búsqueda Lineal

El algoritmo de búsqueda lineal busca un elemento específico en una lista, retornando la posición del elemento en la lista.

Entradas: lista de n números naturales, número a buscar.

Salida: Posición del número en la lista.

Pasos:

1. Comienza con el primer elemento de la lista.
2. Compara el elemento actual con el número buscado.
    a. Si el elemento actual coincide con el número buscado, detén el algoritmo y retorna la posición del elemento actual.
    b. Si el elemento actual no coincide con el número buscado, pasa al siguiente elemento y repite el paso 2.
3. Si ningún elemento de la lista coincide con el buscado, retorna un valor que indique que ninguna posición contiene el elemento buscado (por ejemplo, un -1).

A continuación, se muestra un ejemplo de ejecución de este algoritmo para las entradas [6, 1, 5, 3] (lista de números) y 5 (número a buscar).

1. Comenzamos por el primer elemento, 6:
    - Como $6 \neq 5$, el elemento actual no es el objetivo, avanzamos al siguiente elemento.
2. Comparamos con el siguiente elemento, 1:
    - Como $1 \neq 5$, el elemento actual no es el objetivo, pasamos al siguiente elemento.
3. Comparamos con el siguiente elemento, 5:
    - Como $5 = 5$, encontramos el elemento, detenemos el algoritmo y retornamos la posición del elemento: 2 (se ha considerado que el primer elemento ocupa la posición 0; también podría considerarse que el primer elemento ocupa la posición 1, en cuyo caso el número retornado sería 3).

2.2.3. Algoritmo de Ordenamiento Burbuja (Bubble Sort)

Este algoritmo permite ordenar una lista de números de manera ascendente de manera sencilla.

Entrada: lista de n números enteros.

Salida: lista de n números enteros ordenada.

Pasos:

1. Para cada elemento de la lista (excepto el último):
    a. Compara el elemento actual con el siguiente elemento de la lista.
    b. Si el elemento actual es mayor que el elemento siguiente, intercámbialos.
2. Si se ha realizado algún cambio al recorrer la lista completa, repetir el paso 1.

Se muestra un ejemplo de la ejecución de este algoritmo con la lista de entrada [3, 5, 2]:

1. Cogemos el primer elemento de la lista, comparándolo con el siguiente:
   - Como $3 < 5$, la lista se mantiene igual: [3, 5, 2].
2. Pasamos al segundo elemento de la lista, comparándolo con el siguiente:
   - Como $5 > 2$, los elementos se intercambian, resultando la lista: [3, 2, 5].
3. Al no haber más elementos, terminamos la primera pasada. Sin embargo, al haber realizado cambios durante la misma, es necesario realizar una nueva pasada.
4. Cogemos de nuevo el primer elemento de la lista, comparándolo con el siguiente:
   - Como $3 > 2$, los elementos se intercambian, quedando la lista: [2, 3, 5].
5. Pasamos al segundo elemento, comparándolo con el siguiente:
   - Como $3 < 5$, la lista se mantiene igual: [2, 3, 5].
6. Al no haber más elementos, finalizamos la segunda pasada. Al haber realizado cambios también durante esta pasada, es necesario realizar una nueva pasada.
7. Cogemos el primer elemento de la lista, comparándolo con el siguiente:
   - Como $2 < 3$, la lista se mantiene igual: [2, 3, 5].
8. Pasamos al segundo elemento, comparándolo con el siguiente:
   - Como $3 < 5$, la lista se mantiene igual: [2, 3, 5].
9. Terminamos la tercera pasada, y al no haber realizado cambios no es necesario realizar nuevas pasadas, la lista está ordenada: [2, 3, 5]

3. Lenguajes de programación

Según la RAE, un lenguaje de programación es el conjunto de instrucciones, signos y reglas que permite a los humanos comunicarse con los computadores. Estas instrucciones deben escribirse utilizando una sintaxis que los computadores puedan interpretar y ejecutar, permitiendo crear herramientas software, aplicaciones o sistemas operativos. Existen diferentes clasificaciones de los lenguajes de programación, mostrándose a lo largo de este apartado algunas de ellas.

En función del nivel de abstracción y proximidad al hardware se establecen diferentes tipos de lenguajes de programación:

- Lenguajes de programación de bajo nivel, con una alta proximidad al hardware, haciendo que su velocidad de ejecución sea muy alta y permitiendo al programador gestionar los recursos del hardware. Sin embargo, cuentan con un bajo nivel de abstracción, dificultando la lectura y escritura del código. Además, al utilizar conjuntos de instrucciones tan cercanas al hardware de la máquina, también son altamente dependientes del mismo, siendo necesario modificarlo o reescribirlo para poder ejecutarlo en máquinas de diferentes familias. Encontramos entre este tipo de lenguajes el lenguaje máquina, cuyas instrucciones están en código binario (0s y 1s), y el lenguaje ensamblador, que sustituye el código binario por mnemotécnicos, algo más sencillos de entender para los humanos.
- Lenguajes de programación de alto nivel, los cuales destacan por su alto nivel de abstracción, utilizando sintaxis y estructuras más cercanas al lenguaje humano facilitando su lectura, escritura y depuración. Por otro lado, tienen una baja proximidad al hardware, lo que impide gestionar directamente sus recursos, pero

haciéndolos independientes del mismo, pudiendo ser ejecutados en cualquier máquina sin modificar su código.

- Lenguajes de programación de muy alto nivel, enfocados en resolver determinados problemas, proporcionan una gran abstracción tanto del hardware como de la implementación. Estos lenguajes permiten a los desarrolladores centrarse en la resolución del problema sin pensar en los aspectos técnicos. Dicho de otra forma, estos lenguajes se centran en describir el resultado deseado o problema a resolver en lugar de especificar los pasos para su resolución.

Los lenguajes de programación han ido evolucionando con el tiempo, pudiendo distinguir diferentes generaciones, cada una de las cuales con sus propias características. Usualmente suelen dividirse en las siguientes generaciones:

- Primera generación: código máquina. Consiste en instrucciones en código binario (0s y 1s) que la computadora puede entender directamente, lo que lo ofrece la mayor velocidad de ejecución posible. Sin embargo, debido a su bajo nivel de abstracción, resulta muy difícil de leer, escribir y depurar. Además, existe el inconveniente añadido de que es específico de la arquitectura del procesador, por lo que los programas desarrollados deben adaptarse/reescribirse en función de la máquina donde vayan a ser ejecutados.
- Segunda generación: ensamblador. Utiliza mnemotécnicos y nombres simbólicos en lugar de códigos binarios, facilitando la programación con respecto a la generación anterior. Sin embargo, siguen siendo dependientes de la arquitectura del procesador.
- Tercera generación: lenguajes de alto nivel. Introducen un mayor nivel de abstracción, facilitando la programación al utilizar un lenguaje más cercano al lenguaje humano. Además, son independientes de la arquitectura específica del procesador. Algunos ejemplos de lenguajes de esta generación son C, Java, Python Fortran o Cobol.
- Cuarta generación: lenguajes declarativos y herramientas de desarrollo, también conocidos como lenguajes de muy alto nivel o lenguajes de consulta. Son más abstractos que los lenguajes de la generación anterior, y son específicos para determinadas tareas, como bases de datos o procesamiento de datos. Se enfocan en qué se quiere conseguir en lugar de en cómo hacerlo. Ejemplos de esta generación de lenguajes son SQL, MATLAB o R.
- Quinta generación: lenguajes de programación lógica, también conocidos como lenguajes basados en Inteligencia Artificial o en la resolución de problemas. Estos lenguajes están diseñados para facilitar la creación de sistemas que aprenden y evolucionan utilizando restricciones y lógica. Como ejemplos de esta generación encontramos Prolog y OPS5.

Otra clasificación interesante se basa en los paradigmas de programación. Atendiendo a estos podemos encontrar:

- Lenguajes imperativos: el programador indica las instrucciones que la computadora debe seguir para resolver un problema. Algunos lenguajes imperativos son C, Java y Python.

- Lenguajes declarativos: el programador describe el resultado, es decir, qué debe realizar el programa, sin especificar los pasos exactos para ello. Lenguajes de este tipo son SQL y HTML.
- Lenguajes funcionales: se basan en el uso de funciones matemáticas. Destacan por su inmutabilidad (los datos no cambian de valor una vez creados, en su lugar se crean nuevos datos) y el uso de recursividad en lugar de bucles. Entre estos lenguajes podemos encontrar Haskell y Lisp.
- Lenguajes orientados a objetos: se centran en el uso de clases y objetos para representar datos. Destaca su encapsulamiento y su uso de la herencia. Entre estos lenguajes encontramos Java, C++, Python y Ruby.
- Lenguajes lógicos: utilizan reglas lógicas para expresar relaciones y realizar inferencias. Un ejemplo de este tipo de lenguajes es Prolog.

4. Compiladores e intérpretes

Como ya se ha indicado en el apartado anterior, los computadores únicamente entienden el lenguaje máquina (conjunto de instrucciones binarias), por lo que para poder ejecutar nuestro programa es necesario que se encuentre en este lenguaje. Por ello, si utilizamos un lenguaje de alto nivel es necesario traducir nuestro código a este lenguaje máquina. Esta es la función de los compiladores e intérpretes: traducir el código escrito por los desarrolladores en código que los procesadores puedan entender y ejecutar.

Los compiladores son programas que traducen el código fuente de un lenguaje de programación a lenguaje máquina, generando un archivo que el procesador puede ejecutar (por ejemplo, los archivos .exe en Windows). Por su parte, los intérpretes son programas que van traduciendo y ejecutando el código fuente línea a línea, por lo que no se genera un archivo ejecutable. A diferencia de los compiladores, que traducen todo el programa de una vez para su posterior ejecución, los intérpretes traducen y ejecutan instrucción a instrucción.

Cada enfoque tiene sus propias ventajas e inconvenientes. Así, los compiladores, al analizar y traducir todo el código de una vez permiten detectar errores antes de ejecutar el programa, además de permitir una rápida ejecución (una vez traducido el código es posible ejecutarlo todas las veces que sea necesario). Sin embargo, esta traducción requiere de cierto tiempo, especialmente en programas muy extensos, siendo necesario recompilar nuestro programa cada vez que se realice un cambio. Por su parte, los intérpretes permiten ejecutar el código, depurarlo y modificarlo en tiempo real, ya que una instrucción no se traduce hasta que se llega a ese punto. Sin embargo, la ejecución de los programas interpretados es más lenta (es necesario traducirlos cada vez que se ejecutan), y los errores no detectados pueden provocar paradas inesperadas de nuestro programa.

No existe un enfoque mejor que otro, ya que ambos tienen sus ventajas dependiendo del contexto. Usualmente, los lenguajes compilados (como C, C++ o Java) son ideales para aplicaciones que requieren alto rendimiento, mientras que los programas interpretados (como Python, Ruby o JavaScript) permiten un rápido desarrollo y aportan una gran flexibilidad.

5. Entornos de programación

Según lo visto anteriormente, para comenzar a programar únicamente es necesario un editor de texto donde escribir nuestro programa y un compilador o intérprete que traduzca el código de nuestro programa a código máquina. Sin embargo, existen aplicaciones denominadas IDEs (Integrated Development Environments) que proporcionan un conjunto de herramientas que facilitan la escritura, prueba y depuración de nuestros programas. Existen múltiples IDEs, la mayoría de los cuales permiten ser configurados para utilizarse con diferentes lenguajes de programación. Existen múltiples IDEs, siendo en C los más utilizados Eclipse, Visual Studio, Netbeans, Dev-C++, Qt Creator, CodeBlocks y CLion. En nuestro caso, se propone el uso de CLion, aunque la mayoría son similares e incluyen características comunes. Las principales características se señalan sobre la interfaz de CLion en la Figura 1, detallándose después:

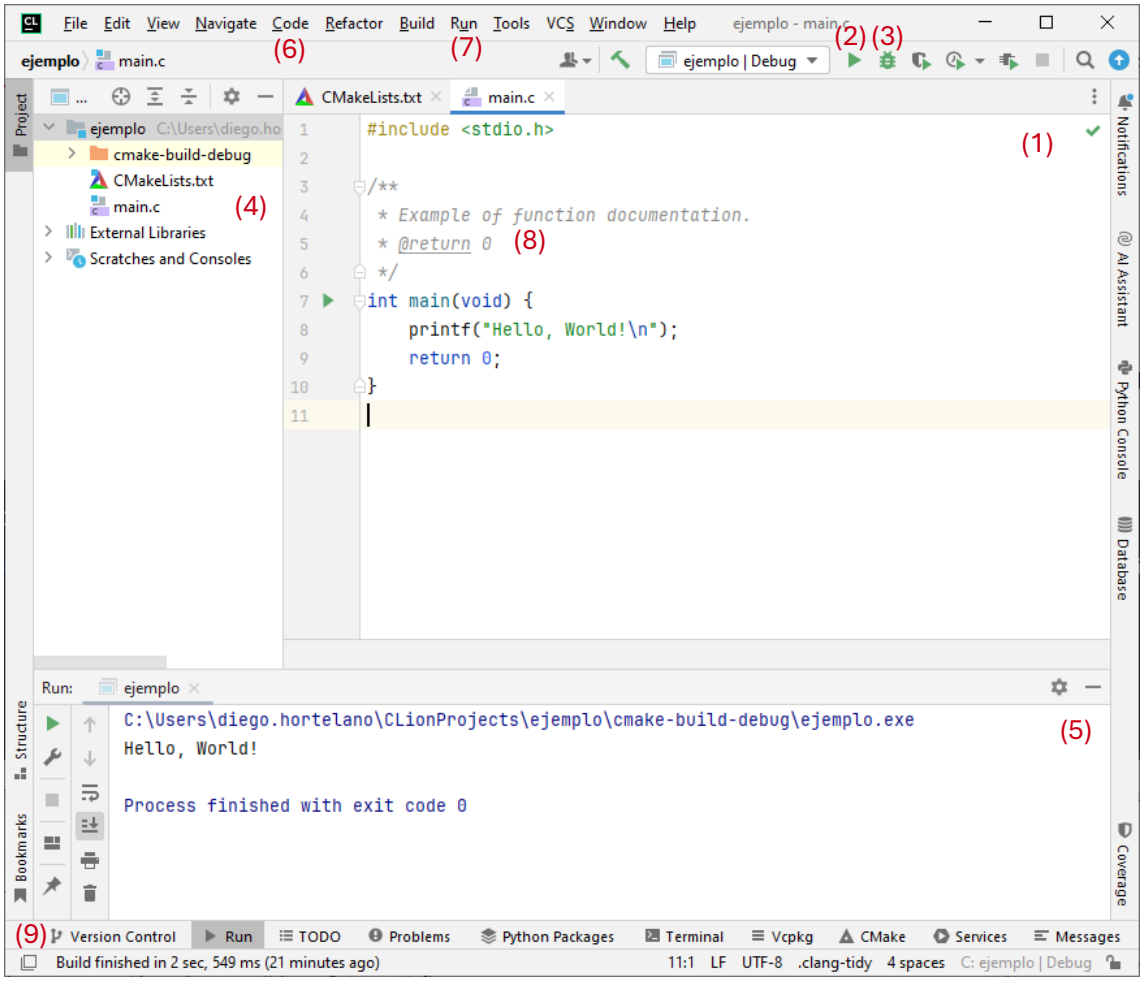


Figura 1. Captura de pantalla de un IDE donde se señalan sus principales herramientas.

Las características detallas en la Figura 1 son:

1. Editor de texto: permite escribir el código de nuestro programa, ofreciendo funciones como resaltar la sintaxis o autocompletar palabras.
2. Compilador o intérprete: permite traducir el código a código máquina simplemente pulsando un botón.
3. Depurador (o debbuger): permite ejecutar nuestro programa paso a paso, lo que ayuda a encontrar y solucionar errores.
4. Gestor de proyectos: facilita la organización de ficheros y directorios dentro de un proyecto de programación.
5. Consola integrada: permite interactuar con la ejecución de nuestro programa, viendo los mensajes de salida o errores o ingresando comandos.
6. Analizador en tiempo de ejecución: estudian la eficiencia de los programas desarrollados.
7. Herramientas de pruebas: nos permiten verificar que nuestro código funciona correctamente.
8. Generadores de documentación: construyen la documentación de nuestro programa a partir de los comentarios incluidos en el mismo.
9. Sistemas de control de versiones: permiten integrar sistemas para gestionar y seguir los cambios realizados en nuestro código a lo largo del tiempo.

6. Pasos para la creación de un programa en C

Como hemos visto, C es un lenguaje de alto nivel que debe ser traducido para ser ejecutado. Esto conlleva una serie de pasos clave, los cuales se presentan a continuación:

1. **Edición**: escritura del programa en un editor de texto o un IDE. En este paso se crea un fichero fuente con extensión .c para C.
2. **Compilación**: traducción del código fuente a un lenguaje intermedio denominado código objeto, que contiene instrucciones en lenguaje máquina pero que no es posible ejecutar. Uno de los compiladores más utilizados para C es gcc (GNU Compiler Collection). Este paso genera un archivo con extensión .obj.
3. **Ensamblado**: este paso se incluye dentro de la compilación, y también es realizado por el compilador. Se trata de la conversión del código objeto en instrucciones máquina específicas para el procesador.
4. **Enlazado**: esta etapa también suele incluirse en la etapa de compilación, y consiste en la combinación del código objeto generado con otras librerías y dependencias necesarias para crear el archivo ejecutable. En estas etapas (ensamblado y enlazado) se genera el archivo ejecutable, cuya extensión depende del sistema operativo. En Windows, son archivos con extensión .exe.
5. **Ejecución**: es el proceso de lanzar nuestro archivo ejecutable. Si todos los pasos se han realizado correctamente y no hay errores, podremos ver la salida de nuestro programa.
6. **Depuración**: es el proceso de encontrar y solucionar errores. Estos errores pueden producirse en las diferentes etapas, encontrando errores de compilación, de ensamblado o de ejecución, siendo estos últimos los más complicados de encontrar y solucionar y donde más valor cobra la depuración. Para depurar nuestro programa podemos hacer uso de un depurador, que nos permitirá ejecutar el

programa paso a paso y establecer puntos de interrupción para detener la ejecución del programa, pudiendo inspeccionar en todo momento el valor de las variables definidas.

7. Pseudocódigo

El pseudocódigo es un lenguaje de descripción de algoritmos de alto nivel que combina lenguaje natural y elementos de los lenguajes de programación para representar algoritmos y funciones de manera compacta. El pseudocódigo permite describir un programa de manera que los desarrolladores (o personas involucradas de alguna forma en el proceso de desarrollo) puedan comprenderlo fácilmente, primando la lectura humana. De hecho, se denomina pseudocódigo porque no es realmente ejecutable. Además, el pseudocódigo es independiente de los lenguajes de programación, aunque puede verse influenciado por ellos, permitiendo comprender la descripción a pesar de que no todos los programadores utilicen el mismo lenguaje. A continuación, se muestran algunas instrucciones en pseudocódigo:

1. Asignar valores:

$$x = y \qquad\qquad x \leftarrow y$$

2. Operaciones matemáticas:

$$res = num1 \cdot num2 \qquad\qquad res = y \,\%\, 3$$

3. Operaciones lógicas:

$$x < 5 \qquad x! = 5 \; AND \; y \geq 2 \qquad y == 1$$

4. Estructuras condicionales:

```
Si condición Entonces
        instrucciones
Fin Si
```

```
Si condición Entonces
        instrucciones
Si no Entonces
        instrucciones
Fin Si
```

5. Estructuras iterativas o de repetición:

```
Mientras condición Hacer
        instrucciones
Fin Mientras
```

```
Repetir
        Instrucciones
Hasta Que condición
```

```
Para var = 1 Hasta n Hacer
        instrucciones
Fin Para
```

Cabe destacar aquí que las estructuras de repetición *Mientras* y *Repetir* pueden intercambiarse si invertimos la condición. Además, es necesario que la condición se modifique en el bloque de instrucciones de la estructura de repetición para que el programa finalice.

7.1. Ejemplos de pseudocódigos

En este apartado se incluyen algunos ejemplos de pseudocódigo de los algoritmos vistos anteriormente. En ambos casos se ha considerado que el primer índice de las listas se encuentra en la posición 0 en lugar de 1 por proximidad a la mayoría de lenguajes de

programación, que utilizan esta convención, pero se podría modificar aplicando las correcciones necesarias.

7.1.1. Algoritmo para extraer el valor mínimo

Un posible pseudocódigo para el algoritmo expuesto en el punto 2.2.1. Algoritmo para extraer el valor mínimo podría ser el siguiente:

```
Algoritmo EncontrarMinimo
        leer lista
        longitud = longitud(lista)
        mínimo = lista[0]
        Para i=1 hasta longitud - 1 Hacer:
                elemento_actual = lista[i]
                Si elemento_actual < mínimo Entonces:
                        mínimo = elemento_actual
                Fin Si
        Fin Para
        escribir mínimo
Fin Algoritmo EncontrarMinimo
```

7.1.2. Algoritmo de Búsqueda Lineal

Un posible pseudocódigo para el algoritmo expuesto en el punto 2.2.2. Algoritmo de Búsqueda Lineal podría ser el siguiente:

```
Algoritmo BúsquedaLineal
        leer lista
        leer número_buscar
        encontrado = falso
        longitud = longitud(lista)
        posición_actual = 0
        Mientras posición_actual < longitud AND encontrado == falso Hacer:
                elemento_Actual = lista[posición_actual]
                Si elemento_Actual == número_buscar Entonces:
                        encontrado = cierto
                Si no Entonces
                        posición_actual = posición_actual + 1
                Fin Si
        Fin Mientras
        Si encontrado == cierto Entonces:
                escribir posición_actual
        Si no Entonces
                escribir "Elemento no encontrado"
        Fin Si
Fin Algoritmo BúsquedaLineal
```

Tema 2: Fundamentos del lenguaje C

1. El lenguaje C

El lenguaje C es uno de los lenguajes más utilizados en la actualidad, situado en la cuarta posición en el índice TIOBE [1] y en la novena en el índice IEEE Spectrum [2]. Ambos rankings tratan de medir la popularidad de los diferentes lenguajes de programación, basándose para en ello en métricas como cuántos ingenieros cualificados en todo el mundo que utilizan un determinado lenguaje de programación, el número de cursos sobre ese lenguaje o el número de ofertas de empleo que requieren dicho lenguaje de programación.

1.1. Historia de C

El nacimiento de C está íntimamente ligado al desarrollo de los sistemas operativos Unix. Esta familia de sistemas operativos deriva del Unix original [3], el cual comenzó su desarrollo en el centro de investigación Bell Labs en 1969, siendo sus principales contribuidores Ken Thompson y Dennis Ritchie. Este sistema operativo estaba siendo desarrollado en ensamblador en un PDP-7[1] [4], por lo que, con el fin de facilitar el desarrollo de aplicaciones software, Thompson creó una versión reducida del lenguaje de programación de sistemas llamado BCPL[2], reduciendo su sintaxis y acercándolo a la sintaxis de SMALGOL[3] [5], adaptándolo a las necesidades de este sistema. Thompson denominó a este nuevo lenguaje B, y lo describió como "semántica de BCPL con mucha sintaxis de SMALGOL" [6].

Posteriormente, con el lanzamiento de PDP-11[4], Dennis Ritchie decidió adaptar el lenguaje B para aprovechar las nuevas características de estos minicomputadores. Por ello, comenzó a mejorar el lenguaje B en 1971, incluyendo el tipado de variables y añadiendo un nuevo tipo de datos, el carácter, denominando a este nuevo lenguaje NB (New B). Por otra parte, el desarrollo del sistema operativo Unix también se migró a PDP-11, y Thompson decidió reescribirlo en un lenguaje de programación de alto nivel, eligiendo para ello NB. Los primeros tres intentos de Thompson de reescribir Unix en un lenguaje de alto nivel terminaron en fracaso, pero permitieron incluir nuevas características en el lenguaje NB, como arrays, punteros y estructuras, lo que permitió finalmente implementar Unix en este nuevo lenguaje. Ritchie y Thompson consideraron

[1] PDP-7 fue un minicomputador de 18 bits lanzado en 1965 y producido por Digital Equipment Corporation (DEC) como parte de la serie DPD (Programmed Data Processor) [12].
[2] BCPL (Basic Combined Programming Language) es un lenguaje de programación implementado por Martin Richards y destinado inicialmente para escribir compiladores para otros lenguajes [13].
[3] SMALGOL es una versión reducida del lenguaje ALGOL (Algorithmic Language), una familia de lenguajes originalmente desarrollada en 1958 y utilizada ampliamente en la descripción de algoritmos [6].
[4] PDP-11 fue una serie de minicomputadores de 16 bits producidos por DEC y que estuvieron en producción desde 1970 hasta 1997 [14].

©2024 Diego Hortelano Haro, Gerardo Reyes Salgado, Manuel Rubio Sánchez. Algunos derechos reservados. Este documento se distribuye bajo la licencia "Atribución/Reconocimiento-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en https://creativecommons.org/licenses/by-sa/4.0/deed.es.

que todas estas nuevas características eran un cambio suficiente para modificar también el nombre del lenguaje de programación, pasando a denominarse C [5].

Durante la siguiente década, C ganó gran popularidad debido a su eficiencia, utilizándose para desarrollar gran cantidad de software, incluyendo el sistema operativo Unix, lo que llevó a su adopción generalizada en la comunidad de desarrolladores. A medida que el uso de C se extendió, surgió la necesidad de un estándar formal para asegurar la portabilidad y consistencia en diferentes sistemas, por lo que, en 1983 fue estandarizado por el American National Standards Institute (ANSI), dando lugar a los sucesivos estándares ANSI C, siendo la primera de ellas la ISO/IEC 9899:1990 (conocida comúnmente como C89 o C90). Posteriormente ha habido diferentes revisiones del estándar, dando lugar a diferentes normas, como la ISO/IEC 9899:1999 (C99), la ISO/IEC 9899:2011 (C11), la ISO/IEC 9899:2018 (C17) o la ISO/IEC 9899:2024 (C23), actualmente en desarrollo.

1.2. Características y utilidad de C

El lenguaje de programación C destaca por las siguientes características:

- Permite producir código **eficiente** y rápido, lo que lo convierte en una alternativa ideal para aplicaciones de alto rendimiento, como sistemas operativos, software de tiempo real, sistemas distribuidos y sistemas empotrados [7].
- El código escrito en C puede compilarse y ejecutarse en diferentes plataformas sin apenas modificaciones, facilitando su **portabilidad** y desarrollo de software **multiplataforma** [8].
- Se considera un lenguaje de **nivel medio**, debido a que combina características de los lenguajes de alto nivel y de bajo nivel. Así, C cuenta con estructuras de control, funciones y bibliotecas estándar que permiten escribir un código legible y manejable, de manera similar a los lenguajes de **alto nivel**. Además, C permite manipular directamente la memoria y los registros del hardware, lo que permite un control preciso sobre el mismo, similar al lenguaje ensamblador (de **bajo nivel**). Esto lo hace especialmente útil en la programación de sistemas y microcontroladores [8].
- Cuenta con un conjunto completo de **instrucciones de control**, como condiciones, bucles y funciones, permitiendo una programación estructurada y modular [9].
- Facilita el **uso de punteros**, los cuales proporcionan un control preciso sobre la gestión de memoria y permitiendo la manipulación arrays y la gestión de memoria dinámica [7].
- Ofrece una amplia gama de **bibliotecas** con funciones estándar para realizar tareas comunes, facilitando el desarrollo de aplicaciones complejas [8].
- Permite dividir el código en **funciones** y **módulos**, facilitando la organización, mantenimiento y reutilización del código [8].
- Ofrece una rica variedad de **tipos de datos** básicos, y permite la creación de tipos de datos complejos como estructuras, uniones o arrays [8].

Todas estas características lo han hecho un lenguaje ampliamente utilizado, destacando en el desarrollo de sistemas operativos, sistemas distribuidos y sistemas empotrados, así

como en redes, telemática y transmisión de datos. Además, es un lenguaje de propósito general utilizado también en otros campos como visión artificial y optimización.

2. Estructura y elementos de un programa en C

Los programas en C tienen una estructura específica que se compone de varios elementos. Entre los principales encontramos:

- Directivas del preprocesador.
- Funciones, incluyendo la función main, obligatoria en todos los programas, ya que es la que se ejecuta cuando lanzamos nuestro programa.
- Declaraciones de variables, de diferentes ámbitos.
- Comentarios.

La Figura 1 muestra la estructura de un programa típico en C, que incluye los elementos mencionados anteriormente. Estos elementos se detallan en los siguientes puntos.

```c
/*
 * Nombre: Código de ejemplo
 * Autor: Diego
 */

/* Directivas del preprocesador */
#include <stdio.h>

/* Definición de viariables globales */
int variable_global;

/* Función que imprime el valor de la variable global y retorna 1. */
int funcion()
{
    // imprimimos el valor de la variable global
    printf("valor variable_global en funcion: %d\n", variable_global);
    return 1;
}

//Función main
int main(void) {
    // Definición de variables locales
    int variable_local = 2;

    variable_global = variable_local;

    // imprimimos por pantalla la variable local y la variable global
    printf("valor variable_global: %d\n", variable_global);
    printf("valor variable_local: %d\n", variable_local);

    funcion();

    return 0;
}
```

Figura 1. Estructura de un programa en C.

3. Comentarios

Los comentarios son líneas que se ignoran por el compilador, es decir, no influyen al funcionamiento del programa. Estas líneas sirven para documentar el código, facilitando su posterior comprensión y mantenimiento, tanto para el desarrollador original como para otros desarrolladores.

En C existen dos tipos de comentarios, dependiendo del número de líneas que ocupen. Los comentarios de una línea se indican con //, y permiten añadir notas breves. Todo lo que se escriba a continuación de // en esa misma línea será ignorado por el compilador. Un ejemplo de este tipo de comentarios se encuentra en la Figura 2.

```
14  int funcion()
15  {
16      printf("valor variable_global en funcion: %d\n", variable_global); // imprimimos el valor de la variable global
17      return 1;
18  }
```

Figura 2. Ejemplo de comentario de una línea.

Por otra parte, los comentarios de múltiples líneas se utilizan para incluir descripciones más detalladas, para comentar bloques de código o para documentar funciones. Estos comentarios comienzan con /∗ y finalizan con ∗/. El compilador ignorará todo lo que se encuentre entre estos símbolos. La Figura 3 muestra un ejemplo de este tipo de comentarios.

```
12  /*
13   * Comentario de múltiples líneas.
14   * Este comentario se utiliza para explicar el funcionamiento
15   * del siguiente bloque de código
16   */
17  int funcion()
18  {
19      printf("valor variable_global en funcion: %d\n", variable_global);
20      return 1;
21  }
```

Figura 3. Ejemplo de comentario de múltiples líneas.

Los comentarios deben ser claros y aportar información, debiendo evitar comentarios innecesarios y que no aporten valor. Una buena práctica es incluir comentarios que expliquen el porqué de una sección del código, no solo el qué.

4. Directivas del preprocesador

Las directivas del preprocesador en C son instrucciones que se ejecutan en la etapa previa a la compilación del código. Estas instrucciones preparan el código fuente antes de la compilación incluyendo archivos o definiendo valores o macros, por ejemplo. Las directivas del preprocesador comienzan por #, y es escriben al comienzo del programa. Entre las principales encontramos:

- #***include***: incluye el contenido de otro archivo en nuestro código fuente. Los archivos incluidos pueden ser bibliotecas (con $\#include < archivo >$, por ejemplo, **`#include <stdio.h>`**) o archivos definidos por el usuario (con $\#include$ "archivo", por ejemplo, **`#include "archivo.h"`**), como veremos en

el Tema 7. Esta directiva permite reutilizar código y organizar nuestros proyectos de manera modular.

- ***#define***: permite definir constantes y macros. Por un lado, podemos utilizar *#define* para definir constantes que pueden utilizarse en el código en lugar de los valores literales, aportando legibilidad. Por ejemplo, `#define PI 3.14` define una constante denominada `PI`, que podremos utilizar en nuestro código, donde se reemplazará por `3.14`. Por otra parte, podemos definir macros que acepten parámetros, facilitando la reutilización de código, como `#define CUADRADO(x) ((x) * (x))`. En este caso, la expresión `CUADRADO(x)` de nuestro código se sustituirá por `((x) * (x))`. A continuación, se muestra un ejemplo con ambas definiciones:

```c
#include <stdio.h>

#define PI 3.14
#define CUADRADO(x) ((x) * (x))

int main() {
    double radio = 5.0;
    double area = PI * CUADRADO(radio);

    printf("El área del círculo es: %f\n", area);
    return 0;
}
```

Este código es totalmente equivalente a:

```c
#include <stdio.h>

int main() {
    double radio = 5.0;
    double area = 3.14 * ((radio) * (radio));

    printf("El área del círculo es: %f\n", area);
    return 0;
}
```

- ***#if***, ***#ifdef***, ***#ifndef***, ***#else***, ***#elif***, ***#endif***: permiten controlar la compilación de bloques de código en función de si se cumplen o no determinadas condiciones. Esto es especialmente útil para manejar el código en función de la configuración elegida o de la plataforma actual. Por ejemplo, el siguiente código solamente se compilaría y, por tanto, ejecutaría, si la variable DEBUG está definida (`#define DEBUG`):

```c
#include <stdio.h>

#define DEBUG   // Definir DEBUG

int main() {
#ifdef DEBUG
    printf("Modo Debug\n");
#endif

    printf("Modo Normal\n");
    return 0;
}
```

- Otras directivas, como ***#undef***, utilizada para eliminar macros y constantes, ***#pragma***, que permite enviar instrucciones al compilador, aunque puede ser necesario modificarlas si cambiamos de compilador, o ***#error***, que genera un mensaje de error durante la compilación.

5. Identificadores

Los identificadores son nombres que podemos asignar a los distintos elementos de un programa, como variables, funciones o tipos, por ejemplo. Los identificadores permiten hacer referencia a estos elementos de manera inequívoca, accediendo a la dirección de memoria donde se encuentran dichos elementos.

Como buena práctica, se recomienda que los identificadores sean claros y descriptivos, para facilitar el mantenimiento y legibilidad de nuestro código. Además, en el lenguaje C hay una serie de reglas que debemos seguir a la hora de elegir un identificador válido (no todos lo son):

- Debe ser una cadena de caracteres, la cual puede contener únicamente letras mayúsculas y minúsculas, dígitos numéricos (0-9) y el guion bajo ('_').
- No puede contener espacios.
- No puede contener signos de puntuación.
- Debe comenzar obligatoriamente por una letra (mayúscula o minúscula) o guion bajo, aunque el uso de estos últimos está reservado para el uso de los compiladores y bibliotecas, por lo que es preferible evitar el uso de estos identificadores.
- Se recomienda que los identificadores no tengan más de 31 caracteres. Aunque algunos compiladores modernos trabajan correctamente con identificadores más largos, el estándar ANSI C no los contemplaba, por lo que, si nuestro código necesita ser compilado por un compilador que siga este estándar es posible que tengamos problemas de compatibilidad.
- Un identificador no puede ser una palabra reservada. Las palabras reservadas de un lenguaje de programación son palabras que tienen un determinado significado dentro del lenguaje. Estas palabras no pueden utilizarse como identificadores porque ya tienen un propósito en el lenguaje. En C hay un total de 32 palabras reservadas: `auto`, `break`, `case`, `char`, `const`, `continue`, `default`, `do`, `double`, `else`, `enum`, `extern`, `float`, `for`, `goto`, `if`, `int`, `long`, `register`, `return`, `short`, `signed`, `sizeof`, `static`, `struct`, `switch`, `typedef`, `union`, `unsigned`, `void`, `volatile`, `while`.

Además, los identificadores distinguen entre mayúsculas y minúsculas, por lo que "`variable`" y "`Variable`" serían dos identificadores válidos diferentes. Es necesario tener esto en cuenta a la hora de utilizar los elementos definidos.

Así, como ejemplos de identificadores válidos podemos tener "`minimo`", "`MAX_DIGIT`", "`area_circulo`", "`numero1`", "`resultadoFinal`" o "`temp123`", mientras que ejemplos de identificadores no válidos serían "`1variable`" (comienza con un dígito numérico), "`resultado-final`" (contiene un guion), "`float`" (es una palabra reservada), "`#maximo`", "`total$`" o "`@usuario`" (contienen un carácter especial).

6. Tipos de datos

En C existen tres tipos de datos básicos, cada uno de los cuales viene definido por su representación en memoria y su rango, así como por las operaciones que pueden realizar. Estos tres tipos son enteros (`int`), reales (`float`) y caracteres (`char`).

6.1. Enteros (int)

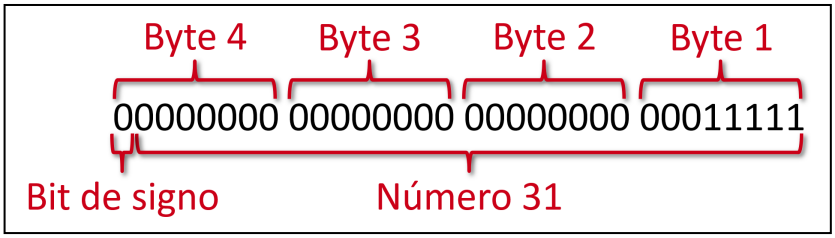
Permiten almacenar números enteros, tanto positivos como negativos. En C, el tamaño que ocupan estos números en memoria depende de la arquitectura del sistema y del compilador, y puede ser 2 o 4 bytes. Aunque en la mayoría de los equipos actuales un entero utiliza 4 bytes de espacio, es necesario tener esto en cuenta para asegurar la compatibilidad de nuestros programas.

Además, este tamaño influye en el rango de números que puede almacenar una variable de este tipo. Así, si disponemos de 2 bytes (o 16 bits) de memoria, podremos representar hasta 2^{16} (65.536) números diferentes, mientras que si la memoria disponible es de 4 bytes (o 32 bits), seremos capaces de representar hasta 2^{32} $(4.294.967.296)$ números. Si tenemos en cuenta que es necesario almacenar tanto números positivos como negativos, obtenemos los siguientes rangos:

- Del -32.768 al 32.767 si los números enteros son de 2 bytes.
- Del $-2.147.483.648$ al $2.147.483.647$ si los números enteros son de 4 bytes.

La cantidad y rango de los números que pueden representarse se debe a la manera en la que se almacenan los números en memoria. En caso de los números enteros positivos, se almacena su representación en binario, dejando el bit más significativo para el signo. Así, para el número 31, tendríamos:

$$(31)_{10} = (11111)_2$$



En lo referente a los números enteros negativos, existen varios modos de representarlos como el Most Significant Byte (que consiste en indicar el signo con el bit más significativo y mantener la representación del número idéntica al número positivo), el Complemento a 1 (en la cual se invierten todos los bits, cambiando los cada 0 por un 1 y cada 1 por un 0), y el Complemento a 2 (el cual se obtiene al tomar el Complemento a 1 de un número y sumarle 1). En C, los números enteros negativos se almacenan utilizando el Complemento a 2, ya que permite una mayor eficiencia en las operaciones de suma y resta. Con todo ello, la representación del número -31 sería:

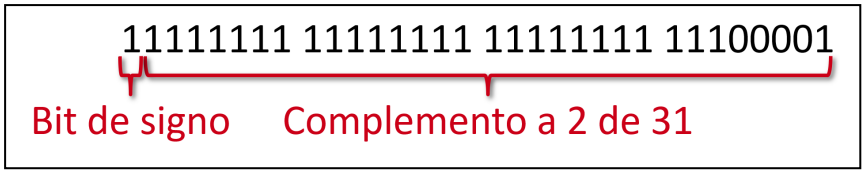
- Pasamos el número 31 a binario, quedando, al igual que antes:

$$00000000\ 00000000\ 00000000\ 00011111$$

- Obtenemos el Complemento a 1, invirtiendo los valores de cada uno de los bits:

$$11111111\ 11111111\ 11111111\ 11100000$$

- Sumamos 1 al Complemento a 1 para obtener el Complemento a 2, quedando:



Además del tipo `int` básico, el lenguaje C incluye una serie de modificadores que permiten cambiar su rango y precisión (así como el tamaño que ocupan en memoria). Estos modificadores se recogen en la siguiente tabla:

Tipo en C	Descripción	Rango	Nº bytes
short int **short**	Entero corto con signo	$-32.768 \dots 32.767$	2
unsigned short int **unsigned short**	Entero corto sin signo	$0 \dots 65535$	2
int	Entero con signo	$-32.768 \dots 32.767$ $-2.147.483.648 \dots 2.147.483.647$	2 4
unsigned int	Entero sin signo	$0 \dots 65535$ $0 \dots 4.294.967.295$	2 4
long int **long**	Entero largo con signo	$-2.147.483.648 \dots 2.147.483.647$	4
unsigned long int **unsigned long**	Entero largo sin signo	$0 \dots 4.294.967.295$	4

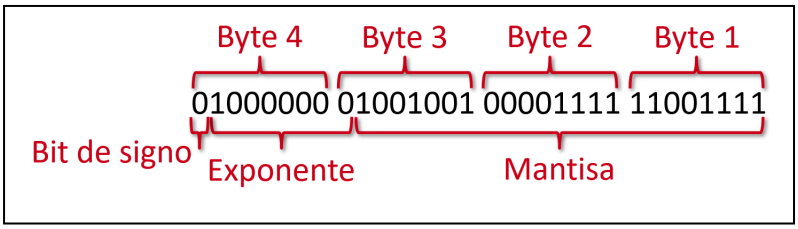
6.2. Reales (float)

Permiten almacenar números decimales, tanto positivos o negativos. Los números reales en C también se almacenan ocupando 4 bytes. Sin embargo, lo hacen siguiendo el estándar para aritmética en punto flotante IEEE 754 [10], transformando el número a una potencia de dos y codificándolo posteriormente con mantisa y exponente. Esto permite representar números en el rango de $-3.4 \cdot 10^{38}$ a $3.4 \cdot 10^{38}$. Para ello, se siguen los siguientes pasos:

1. Convertir el número decimal a binario.
2. Normalizar el número, de manera que obtengamos un número con la forma $1.XXXX \cdot 2^{exp}$.
3. Determinar el exponente sesgado. El exponente real vendrá dado por el exponente del número normalizado (que equivaldrá al número de dígitos que hemos desplazado el número binario). A este exponente, que puede ser positivo o negativo, es necesario sumarle el sesgo (127 para la precisión simple), de manera que permita la representación de exponentes positivos y negativos (en el rango de -126 a 127) sin necesidad de incluir el signo.
4. Codificar el número, utilizando 1 bit para el signo, 8 bits para el exponente y 23 bits de mantisa o fracción (la parte decimal del número normalizado).

Así, para el número 3.141590, tendríamos:

1. Pasando el número $(3.141590)_{10}$ a binario obtendríamos:
   $$(0011.00100100001111100111110)_2.$$
2. Normalizando el número obtenido, obtendríamos:
   $$(1.00100100001111100111110)_2 \cdot 2^1$$
   Por lo que el exponente real será 1.
3. El exponente sesgado será en este caso $1 + 127 = 128$.
4. Codificando el número tendríamos en primer lugar el bit de signo 0, ya que el número es positivo, seguido del exponente en binario $(128)_{10} = (10000000)_2$, y, por último, la mantisa o fracción (cogiendo los primeros 23 dígitos) 10010010000111111001111. Así, el número en formato IEEE 754, tal y como lo almacena C, quedaría:



Al igual que antes, el tipo `float` en C también tiene una serie de modificadores que permiten cambiar su precisión, rango y tamaño ocupado en memoria. Estos modificadores amplían el número de bytes dedicados a almacenar el dato, aumentando la mantisa y el exponente, permitiendo aumentar tanto el rango (mayor exponente) como la precisión del dato (más decimales significativos). Los modificadores disponibles para el tipo `float` se muestran en la siguiente tabla:

Tipo en C	Descripción	Rango	Nº bytes
float	Real	$-3{,}4 \cdot 10^{38} \dots 3{,}4 \cdot 10^{38}$	4
double	Real con doble precisión	$-1{,}7 \cdot 10^{308} \dots 1{,}7 \cdot 10^{308}$	8
long double	Real de precisión extendida	$-3{,}4 \cdot 10^{4932} \dots 3{,}4 \cdot 10^{4932}$	16

Como se ha visto, este tipo de datos permite representar un gran rango de números, pero, por su naturaleza, tiene el inconveniente de que no puede representar todos los números con precisión, presentando errores en el almacenamiento de ciertos números o errores de redondeo que pueden acumularse si realizamos múltiples cálculos. Es necesario ser conscientes de estas limitaciones a la hora de utilizar estos números, especialmente a la hora de realizar comparaciones. En estos casos donde es necesario realizar comparativa es crucial el uso de un épsilon (ε).

Se conoce como épsilon el valor más pequeño que produce un valor diferente cuando se suma a un número en la representación de punto flotante. Dicho de otra forma, es la diferencia mínima detectable entre dos números representados en punto flotante. Este concepto tiene especial importancia debido a los posibles errores de redondeo introducidos por esta forma de almacenar datos, permitiendo determinar si dos números

son lo suficientemente cercanos para ser considerados iguales. Así, es común considerar si dos números a y b son iguales si se cumple que:

$$|a - b| < \varepsilon$$

En C podemos considerar diferentes valores de épsilon en función de los decimales que nuestra aplicación deba manejar. Otra opción es utilizar los valores definidos por la biblioteca `<float.h>` (que debemos incluir con `#include <float.h>`). Esta biblioteca define los siguientes valores de épsilon, en función del tipo de datos utilizado:

Constante	Precisión	Valor aproximado
FLT_EPSILON	float	$1{,}19 \cdot 10^{-7}$
DBL_EPSILON	double float	$2{,}22 \cdot 10^{-16}$
LDBL_EPSILON	long double	$1{,}08 \cdot 10^{-19}$

6.3. Caracteres (char)

Este tipo se utiliza para almacenar caracteres individuales. Este tipo de datos ocupa 1 byte (8 bits) de memoria, almacenando valores enteros de 0 a 255 (`unsigned char`) o de -128 a 127 (`char` o `signed char`).

El valor entero almacenado en este tipo de dato representa un carácter ASCII (American Standard Code for Information Interchange). Este es un código de caracteres que utiliza 7 bits para representar de manera individual a cada uno de los caracteres, lo que permite identificar hasta 128 caracteres. Más adelante se definió el código ASCII extendido, el cual utiliza 8 bits, lo que permite hacer referencia a otros 128 caracteres diferentes (lo que hace un total de 256). Existen diferentes conjuntos de caracteres de ASCII extendidos, ya que los diferentes idiomas han creado sus propios conjuntos para representar sus caracteres especiales, aunque actualmente no están estandarizados, puede resultar de especial interés la página de código 437 o la codificación ISO-8859-1 o latin1, que contienen los caracteres de los idiomas de Europa occidental.

Por ello, si queremos utilizar estos caracteres en nuestras aplicaciones debemos configurar la codificación de nuestros archivos de código (en CLion, File > File Properties > File Encoding) y la de nuestra consola (File > Settings > Editor > General > Console > Default Encoding) en IBM437. En [11] puede consultarse la tabla ASCII de 7 bits como múltiples extensiones de esta, incluyendo la página de código 437.

Una de las ventajas de que C almacene los caracteres como enteros es que podemos realizar operaciones aritméticas sobre ellos. Así, si tenemos el carácter 'B' (con un valor ASCII de 66) y le sumamos 1, obtendríamos el valor 67, o el carácter equivalente a ese valor, en este caso 'C'. Nótese que los caracteres literales en C se encuentran entrecomillados con comillas simples.

6.4. Datos lógicos (booleanos)

En el lenguaje C no existen los datos lógicos como tal (no hay valores true o false), por lo que se hace uso de una convención para representar esto. En dicha convención el valor 0 se considera falso (false), mientras que cualquier valor diferente de 0 se considera verdadero (o true).

Aunque la opción anterior es la más utilizada, a partir del estándar C99 es posible utilizar los valores `true` y `false`, siendo necesario incluir la biblioteca **`<stdbool.h>`**.

7. Variables (definición, declaración e inicialización).

Las variables son espacios de almacenamiento en la memoria del ordenador, la cual tiene un nombre asociado (identificador) que nos permite hacer referencia a la misma y un valor que puede cambiar durante la ejecución del programa. Una variable también puede verse como un contenedor en memoria con una etiqueta que nos permite almacenar datos para luego acceder a ellos o modificarlos. En C, una variable viene dada por:

- Identificador: nombre de la variable que nos permite hacer referencia a la misma. Este debe ser único y seguir las reglas de los identificadores especificadas en el punto 5 de este documento.
- Tipo de Datos: define el tipo de datos que almacena variable, así como la forma de almacenarlo, tal y como se ha visto en el punto 6 de este documento.
- Valor: dato almacenado en la variable, que puede modificarse durante la ejecución del programa.

7.1. Declaración

Para poder utilizar una variable es necesario haberla declarado previamente. Este proceso reserva la memoria necesaria para la variable, en función del tipo de dato especificado, por lo que es necesario indicar al compilador toda la información necesaria (tanto el tipo de dato como el identificador de la variable declarada). Algunos ejemplos de declaración de variables son:

```
int longitud;
unsigned short edad;
float resultado;
double promedio;
char letra;
```

Lo que declara 5 variables de diferentes tipos, reservando el espacio necesario en memoria. Nótese que la variable `longitud`, al ser de tipo `int`, se reservarán 2 o 4 bytes, dependiendo de la arquitectura del sistema y del compilador.

7.2. Inicialización

Al declarar una variable en el lenguaje de programación C, ésta tendrá un valor desconocido, conteniendo el dato o datos que haya en esa dirección de memoria previamente (memoria basura). Por ello, nunca debemos asumir que una variable tiene inicialmente el valor 0 (o cualquier otro valor), ya que dependerá de la dirección de memoria asignada para esa ejecución y del uso anterior de la memoria.

Por ello, resulta de especial importancia el proceso de inicialización de variables, en el cual se le asigna un valor inicial a la misma. Para ello, es suficiente con asignar un valor en el momento de su declaración, tal y como se muestra a continuación:

```
 int longitude = 153;
unsigned short edad = 37;
float resultado = 0.0f; // La f indica que se trata de un float.
```

```
double promedio = 2.3;
char letra = 'C';  // Los caracteres entre comillas simples.
```

No siempre es obligatorio (aunque sí recomendable) inicializar las variables, pero debemos asegurarnos de que nuestro programa no acceda al valor de una variable antes de almacenar en ella un valor válido.

7.3. Tipos de variables

En C existen tres tipos de variables (aunque también puede contemplarse únicamente como dos), en función del lugar donde se declaren:

- Variables globales: se declaran fuera de todas las funciones, idealmente tras las directivas del preprocesador, y son accesibles desde cualquier parte del programa. Aunque el uso de este tipo de variables puede simplificar el código, se recomienda limitar su uso y optar por variables locales siempre que sea posible, ya que el uso de variables globales dificulta el mantenimiento, depuración y modificación de los programas.
- Variables locales: se declaran dentro de una función, siendo accesibles únicamente desde la función donde están declaradas. Una vez finaliza la ejecución de la función el espacio de memoria reservado para la variable se libera, permitiendo optimizar los recursos.
- Variables de bloque: se declaran dentro de un bloque de código (como estructuras condicionales o de repetición, que se verán más adelante), siendo accesibles únicamente desde dentro del propio bloque. Una vez que el bloque termina su ejecución, el espacio de memoria de la variable se libera.

8. Tamaño de los tipos (sizeof)

Como se ha visto, el número de bytes que ocupa cada tipo de dato en memoria no es totalmente fijo (por ejemplo, el tipo `int` depende de la arquitectura y del compilador). Por ello, con el fin de hacer nuestros programas lo más portables posibles, se recomienda el uso de `sizeof()`. Este operador permite conocer el tamaño exacto que ocupa en memoria un tipo de datos o una variable. La sintaxis de este operador requiere indicar entre sus paréntesis el tipo de dato o el identificador de la variable cuyo tamaño deseamos conocer, retornando el número de bytes que ocupa:

```
int longitud = 20;
sizeof(longitud);  // Retornará 2 o 4 bytes.
sizeof(int);       // Retornará el mismo valor, ya que ambos son int.
```

9. Constantes

Las constantes son valores que no cambian durante la ejecución del programa. Se utilizan para representar valores fijos, utilizándose para mejorar la legibilidad y mantenimiento del código. En C existen dos tipos de constantes:

- Constantes literales, cuyo valor fijo se escribe directamente en el código, como 412, 3.14, 'D' o "Cadena". Nótese que una cadena de caracteres (que se verá en profundidad en el Tema 5) se define utilizando comillas dobles en lugar de comillas simples.

- Constantes simbólicas, las cuales pueden definirse utilizando la directiva del preprocesador `#define` (por ejemplo, `#define PI 3.14`) o con la palabra clave `const`, en cuyo caso se define de forma similar al resto de variables (por ejemplo, `const int PI = 3.14;`). Al utilizar la palabra clave `const` el valor de la variable no podrá modificarse durante la ejecución del programa.

El uso de constantes simbólicas permite mejorar la legibilidad del código y evitan errores de escritura al evitar la repetición del valor. Además, si necesitamos modificar el valor definido únicamente será necesario cambiarlo en la declaración.

10. Lectura y escritura de datos

A la hora de realizar nuestros programas, la lectura y escritura en C son fundamentales para interactuar con los usuarios. Destaca aquí la biblioteca **`<stdio.h>`**, que proviene de *standard input/output,* que deberemos incluir cuando necesitemos utilizar sus funciones. Esta biblioteca ofrece funciones para la lectura de datos por teclado y la escritura de datos por pantalla.

10.1. Códigos de control

Los códigos de control permiten especificar el tipo de los datos en ciertas funciones de lectura y escritura, por lo que encontramos diferentes códigos para cada tipo de dato en C. Los principales son:

Carácter	Nombre
%i	Entero
%u	Entero sin signo
%d	Entero en base decimal
%x	Entero en base hexadecimal
%f	Número en punto flotante
%lf	Flotante de doble precisión
%Lf	Datos de tipo long double
%c	Carácter
%s	Cadena de caracteres
%h %hu	Entero corto y con y sin signo
%ld %lu	Entero largo con y sin signo
%e	Notación científica con e para el exponente
%E	Notación científica con E para el exponente
%Le %LE	Datos de tipo long double con notación científica
%p	Puntero

Nótese que, si el tipo de dato especificado con el código de control no coincide con el tipo de la variable, C intentará convertirlo en el tipo especificado.

10.2. Secuencias de escape

Las secuencias de escape son combinaciones de caracteres que representan caracteres especiales que no es posible indicar de otra forma dentro de una cadena de caracteres. Suelen utilizarse especialmente en la salida. En C se utilizan principalmente:

Carácter	Nombre
\n	Nueva línea
\t	Tabulación
\r	Retorno de carro al inicio de la línea actual
\b	Retroceso
****	Barra invertida ('\')
\'	Comilla simple
\"	Comilla doble
\0	Carácter nulo

10.3. Funciones de salida

Una de las funciones de salida más utilizadas en C es `printf()`, incluida en la biblioteca `<stdio.h>`. Esta función destaca por su versatilidad para mostrar datos con formato por consola, permitiendo mostrar cadenas de caracteres, variables y valores numéricos con el formato deseado y su nombre significa "*print formatted*".

Esta función permite mostrar texto junto con nuestras variables, para lo que es necesario definir una cadena de control que especifique qué variables se imprimirán y cómo lo harán. Para ello se utilizan los códigos de control y las secuencias de escape descritos antes.

La función `printf()` tiene la siguiente sintaxis:

```
printf("Cadena de control", dato1, dato2, ..., datoN);
```

La cadena de control contiene el texto que deseamos incluir. Entre dicho texto se pueden incluir caracteres de escape si queremos hacer uso de ellos en el texto que se mostrará al usuario. Además, es necesario incluir el tipo de los datos que queremos mostrar, así como su formato. Para ello se utilizan los códigos de control indicados anteriormente. Además, es posible especificar el tamaño de cada variable (su número de dígitos) entre el % y el formato (se aplicará alineación izquierda si el número es negativo). Por ejemplo, `%4d` mostrará un número entero decimal con cuatro espacios (siempre que sea posible y no ocupe más espacio) alineado a la derecha. De igual manera es posible indicar el número de dígitos decimales de los números reales, incluyendo el carácter punto ('.') seguido del número de cifras decimales que deseamos entre el carácter % y el formato. Por ejemplo, `%.3f` mostrará un número decimal con tres decimales. A continuación se muestran algunos ejemplos:

```
int i = 516;
float x = 1024.251;
printf(":%6d: \n", i);    // se escribe                    -> :  516:
printf(":%-6d: \n", i);   // alineación izq                -> :516  :
printf(":%2d: \n", i);    // no cabe, formato por defecto  -> :516:
printf(":%12.4f:\n", x);  // anchura 12 con 4 decimales    -> :   1024.2510:
printf(":%12e: \n", x);   // anchura 12 caracteres         -> :1.024251e+03:
printf(":%12.4e:", x);    // anchura 12 con 4 decimales    -> :  1.0243e+03:
```

Además de `printf()`, existen otras funciones de salida, como `puts()` o `fputs()` para mostrar texto (cadenas de caracteres) seguido de un salto de línea por consola o `putchar()` para imprimir un único carácter por consola. A continuación se muestra un ejemplo del uso de estas funciones:

```
puts("Texto a imprimir");
fputs("Texto a imprimir", stdout);
putchar('A');
```

En el caso de la función `fputs()` es necesario indicar el fichero o buffer en el que queremos escribir. En el caso de querer hacerlo por consola debemos indicar el buffer de salida estándar (`stdout`), del cual hablaremos más adelante.

10.4. Funciones de entrada

Por su parte, `scanf()` es una de las funciones de entrada más utilizadas en C para leer de teclado, y se encuentra también en la biblioteca `<stdio.h>`. Al igual que `printf()`, nos permite especificar el formato de la entrada, y su nombre proviene de "*scan formatted*". Esta función interpreta la entrada según el formato especificado, almacenando los datos en las direcciones de memoria (variables) proporcionadas.

Al igual que pasaba con la salida, es necesario especificar el formato de entrada, para lo que es necesario definir una cadena de control y el tipo de datos que queremos almacenar. Para ello, se hace uso de los códigos de control y las secuencias de escape antes descritos.

La función `scanf()` tiene la siguiente sintaxis:

```
scanf("Cadena de control", &var1, &var2, ..., &varN);
```

Al igual que sucede con `printf()`, la cadena de control contiene el tipo de datos y su formato o anchura de las datos que queremos leer y almacenar.

Sin embargo, en este caso las variables que no sean de tipo cadenas de caracteres deben ir precedidas del carácter **`&`**. Este es el operador de direccionamiento, y nos permite obtener la dirección de la variable que se encuentra a continuación, ya que para almacenar los datos es necesario el uso de la dirección de memoria. Sobre esto se profundizará mucho más en el Tema 6. Algunos ejemplos del uso de `scanf()` son:

```
scanf("%d", &varInt);
scanf("%f", &varFloat);
scanf("%d %d %d", &varInt1, &varInt2, &varInt3);
```

Destaca como caso particular el caso de las cadenas de caracteres. Aunque se profundizará sobre su uso en el Tema 5, pueden resultar de gran utilidad para leer o escribir palabras. Para definir una cadena de caracteres es necesario definir un array o lista, en el que debemos indicar el número de elementos de la lista utilizando corchetes (`[]`) a la hora de declarar la variable. Como se ha indicado ya, el código de control para una cadena de caracteres es `%s`, y no requiere incluir el operador `&` para su lectura. Así, podríamos hacer un pequeño programa que lea e imprima una palabra de la siguiente manera:

```
int main()
{
    char palabra[10];
    scanf("%s", palabra);
    printf("%s", palabra);
    return 0;
}
```

Cuando utilizamos `%s` para leer una palabra con `scanf()` se lee hasta que se encuentra algún tipo de espacio (espacio, tabulación o salto de línea). Si queremos leer más de una palabra (es decir, que se incluyan los espacios) podemos utilizar el especificador de formato `[^\n]`, que permite leer hasta alcanzar un salto de línea, quedando la sentencia `scanf("%[^\n]", frase)`.

La biblioteca `<stdio.h>` incluye también otras funciones de entrada, como `getchar()`, que permite leer un carácter, y `gets()` o `fgets()`, que permiten leer cadenas de caracteres hasta un salto de línea. De estas dos últimas, `fgets()` se considera más segura que `gets()`, ya que permite especificar el tamaño máximo del buffer para evitar desbordamientos (de hecho, `gets()` ha sido eliminada de las últimas versiones). A diferencia de `scanf()`, `gets()` y `fgets()` únicamente permiten leer cadenas de caracteres, que deberemos transformar posteriormente a los diferentes tipos de datos. A continuación se muestra un ejemplo de la sintaxis de estas funciones:

```
char c;
char palabras[20];
c = getchar();
gets(palabras');
fgets(palabras, 20, stdin);
```

Como puede observarse, `getchar()` retorna el carácter leído para almacenarlo en una variable de tipo `char`. Por su parte, `gets()` requiere una cadena de caracteres donde almacenar la línea leída. Finalmente, `fgets()` requiere tres parámetros: la cadena de caracteres donde almacenar la línea leída, el número máximo de caracteres a leer y el fichero de lectura. En caso de querer leer de teclado, podemos cambiar el fichero por el buffer de entrada estándar (`stdin`), el cual se detalla en el siguiente punto. En la función `fgets()` podemos hacer uso del operador `sizeof()`, para utilizar como número máximo de caracteres el tamaño de la cadena, de la siguiente manera:

```
char palabras[20];
fgets(palabras, sizeof(palabras), stdin);
```

10.5. Buffers de entrada y salida

En C, los datos se manejan como buffers o streams de bytes. Estos buffers son áreas de memoria temporales que se utilizan para almacenar los datos mientras se transfieren entre el programa y los dispositivos de entrada/salida. Esto permite hacer las operaciones de lectura y escritura de manera eficiente, ya que nuestro programa evita tener que interactuar directamente con los dispositivos de entrada/salida, lo que sería mucho más lento.

Los buffers de entrada se utilizan para almacenar los datos que se leen desde los dispositivos de entrada (como el teclado), mientras que los buffers de salida almacenan los datos que se envían a los dispositivos de salida (como la consola). C proporciona tres buffers estándares (incluidos en la librería `<stdio.h>`). Estos son:

- `stdin`: entrada estándar, usualmente desde el teclado.
- `stdout`: salida estándar, usualmente la consola/pantalla.
- `stderr`: salida de error estándar, utilizada para mostrar mensajes de error.

Gracias a estos buffers podemos hacer uso de funciones como `fgets()` o `fputs()`, que permiten tanto la lectura y escritura en ficheros (de ahí la f en ambas funciones) como la lectura desde la entrada estándar y la salida hacía la salida.

Referencias

[1] TIOBE Software BV, «TIOBE Index for September 2024,» Septiembre 2024. [En línea]. Available: https://www.tiobe.com/tiobe-index/.

[2] S. Cass, «IEEE Spectrum: The Top Programming Languages 2024,» 22 Agosto 2024. [En línea]. Available: https://spectrum.ieee.org/top-programming-languages-2024.

[3] M. D. Mcllroy, «A Research UNIX Reader: Annotated Excerpts from the Programmer's Manual, 1971-1986,» AT&T Bell Laboratories Murray Hill, New Jersey, 1987.

[4] E. S. Raymond, «Origins and History of Unix, 1969-1995,» de *The Art of Unix Programming*, 2003.

[5] R. Jensen, «"A damm stupid thing to do" - the origins of C.,» 12 Septiembre 2020. [En línea]. Available: https://arstechnica.com/features/2020/12/a-damn-stupid-thing-to-do-the-origins-of-c/.

[6] J. Backus, F. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. Perlis, H. Rutishauser, K. Samuelson, B. Vauquois, J. Wegstein, A. van Wijngaarden y M. Woodger, «Revised report on the algorithmic language ALGOL 60,» *Commun. ACM,* vol. 6, nº 1, pp. 1-17, 1963.

[7] B. W. Kernighan y D. M. Ritchie, The C Programming Language, Prentice Hall Professional Technical Reference, 1988.

[8] T. Bailey, An Introduction to the C Programming Language and Software Design, 2005.

[9] M. Olsson, C Quick Syntax Reference, Apress, 2015.

[10] IEEE, «IEEE Standard for Floating-Point Arithmetic,» *IEEE Std 754-2019 (Revision of IEEE 754-2008),* pp. 1-84, 2019.

[11] Injosoft AB, «ASCII Table - Reference to ASCII Table of Windows-1252,» 2024. [En línea]. Available: https://www.ascii-code.com/.

[12] Soemtron, «The Digital Equipment Corporation PDP-7,» [En línea]. Available: https://www.soemtron.org/pdp7.html.

[13] IEEE Computer Society, «Martin Richards, Award Recipient,» 2003. [En línea]. Available: https://www.computer.org/profiles/martin-richards.

[14] D. M. Ritchie, «The development of the C language,» *ACM SIGPLAN Notices,* vol. 28, nº 3, p. 201–208, 1993.

Tema 3: Operadores y Expresiones

En este tema se presentan una serie de conceptos básicos relacionados con los operadores y expresiones de uso más común en el lenguaje C.

1. Instrucción de asignación

Una "instrucción de asignación" en el lenguaje de programación C permite almacenar el valor de una expresión en una variable y que reemplaza el valor que tuviera antes la variable. La estructura básica de una instrucción de asignación es:

$$\boldsymbol{variable} = \boldsymbol{expresión};$$

Donde:

- ***variable***, es un identificador que hace referencia a un espacio de memoria.
- ***expresión***, es una combinación de valores, variables, operadores, o funciones que devuelve un resultado.

La asignación es una de las operaciones más fundamentales en programación y se utiliza para actualizar el valor almacenado en una variable. A continuación se muestra un ejemplo básico de una asignación, donde la variable `x` almacena el valor entero 5:

```
int x;
x = 5;      // Se asigna el valor 5 a la variable x
```

1.1. Operadores de asignación en C

En C, además del operador básico de asignación =, existen otros operadores compuestos de asignación que realizan una operación y asignan el resultado en una sola expresión. A continuación, se describen los operadores de asignación más comunes y sus ejemplos.

1.1.1. Asignación simple (=)

Este operador asigna el valor de la expresión a la izquierda en la variable de la derecha. Por ejemplo, podemos asignar el valor `10` a la variable `a`:

```
int a = 10;
```

1.1.2. Asignación con suma (+=)

Asigna a la variable la suma de su valor actual y el valor de la expresión a la derecha. El siguiente ejemplo asigna a `a` su valor actual más `3`:

```
int a = 5;
a += 3;     // Equivalente a: a=a+3; ahora el valor de a es 8
```

1.1.3. Asignación con resta (−=)

Similar al operador +=, pero en lugar de sumar, resta el valor de la derecha. El siguiente ejemplo asigna a `b` su valor actual menos `4`:

```
int b = 10;
b -= 4;     // Equivalente a: b=b-4; ahora el valor de b es 6
```

©2024 Diego Hortelano Haro, Gerardo Reyes Salgado, Manuel Rubio Sánchez. Algunos derechos reservados. Este documento se distribuye bajo la licencia "Atribución/Reconocimiento-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en https://creativecommons.org/licenses/by-sa/4.0/deed.es.

1.1.4. Asignación con multiplicación (=)*

Multiplica el valor actual de la variable por el valor de la expresión a la derecha. El siguiente ejemplo asigna a `c` su valor actual multiplicado por `2`:

```
int c = 7;
c *= 2;     // Equivalente a: c=c*2; ahora el valor de c es 14
```

1.1.5. Asignación con división (/=)

Divide el valor actual de la variable por el valor de la expresión a la derecha. El siguiente ejemplo asigna a `d` su valor actual dividido entre `4`.

```
int d = 20;
d /= 4;     // Equivalente a: d=d/4; ahora el valor de d es 5
```

1.1.6. Asignación con módulo (% =)

Calcula el resto de la división de la variable entre el valor de la expresión a la derecha. El siguiente ejemplo asigna a `e` el valor del resto de la división de `e` entre `3`:

```
int e = 10;
e %= 3;     // Equivalente a: e=e%3; ahora el valor de e es 1
```

1.2. Ejemplo completo de los principales operadores de asignación

A continuación, se presenta un ejemplo que muestra diferentes operadores de asignación en acción:

```
#include <stdio.h>

int main() {
      int x = 10;
      int y = 5;

      // Asignación simple:
      x = y; // x ahora es igual a 5

      // Asignación con suma:
      x += 2;  // x ahora es 7 (5 + 2)

      // Asignación con resta:
      x -= 3;  // x ahora es 4 (7 - 3)

      // Asignación con multiplicación:
      x *= 2;  // x ahora es 8 (4 * 2)

      // Asignación con división:
      x /= 4;  // x ahora es 2 (8 / 4)

      printf("Resultado final: %d\n", x);       // Salida: 2
      return 0;
}
```

El uso de operadores compuestos de asignación permite escribir código más conciso y eficiente. En C, los operadores de asignación son fundamentales para manipular datos y variables de forma dinámica en los programas. Además, combinarlos con operadores aritméticos o lógicos puede ayudar a simplificar operaciones frecuentes.

2. Expresiones

Una expresión en el lenguaje C es cualquier combinación de ***operandos*** (valores o variables) y ***operadores*** que se evalúa y produce un resultado. Las expresiones son las piezas fundamentales de un programa, ya que representan el procesamiento de los datos, porque definen qué hacer con los operandos para obtener un resultado.

Pueden contener paréntesis para agrupar operaciones y espacios en blanco para mejorar la claridad. A continuación, se muestra un ejemplo básico:

```
int x = 10;
int y = 5;
int z = x + y;    /* La expresión x+y se evalúa y se asigna a
                     la variable z.*/
```

En este ejemplo, `x + y` es una expresión que suma dos operandos y devuelve un valor que luego se asigna a `z`.

3. Tipos de operadores en C

Los operadores pueden ser un carácter o un grupo de caracteres que actúan sobre variables para realizar una operación y obtener un resultado. En C existen operadores aritméticos, incrementales, relacionales y lógicos. Además, atendiendo al número de operandos, los operadores también pueden catalogarse en unarios, binarios, ternarios, etc.

3.1. Operadores aritméticos

Los operadores aritméticos permiten realizar operaciones como la suma (**+**), resta (**–**), multiplicación (*****), división (**/**) y módulo (**%**).

```
int a = 10;
int b = 20;
int result = (a + b) * 2;     // Expresión aritmética.
```

En esta expresión, se realiza una suma entre *a* y *b* utilizando el *operador* + y el resultado se multiplica por 2, para ello se usa el *operador* *. El resultado de esta expresión se almacena en la variable `result`.

3.2. Operadores incrementales

Los operadores incrementales en el lenguaje C permiten aumentar o disminuir el valor de una variable de manera sencilla y eficiente. Estos operadores son ++ (incremento) y −− (decremento), y pueden ser utilizados en modo prefijo o modo sufijo, dependiendo de si se desea incrementar o decrementar el valor antes o después de su uso.

3.2.1. Operador de incremento (++)

El operador ++ se utiliza para aumentar en 1 el valor de una variable. Puede ser colocado antes o después de la variable:

- Prefijo: ++x (incrementa la variable y luego la utiliza).
- Sufijo: x++ (utiliza la variable y luego la incrementa).

A continuación, se incluye un ejemplo del uso de operadores incrementales:

```c
#include <stdio.h>

int main() {
    int x = 5;
    // Incremento prefijo:
    int y = ++x;  // x se incrementa primero, luego se asigna a y
    printf("Incremento prefijo: x = %d, y = %d\n", x, y);
    // Incremento sufijo:
    int z = x++;  // x se asigna a z primero, luego se incrementa
    printf("Incremento sufijo: x = %d, z = %d\n", x, z);
    return 0;
}
```

3.2.2. Operador de decremento (--)

El operador `--` se utiliza para disminuir en 1 el valor de una variable. Al igual que el incremento, puede usarse en modo prefijo o sufijo:

- Prefijo: `--x` (disminuye la variable y luego la utiliza).
- Sufijo: `x--` (utiliza la variable y luego la disminuye).

A continuación, se incluye un ejemplo del uso de operadores incrementales:

```c
#include <stdio.h>

int main() {
    int x = 5;
    // Decremento prefijo:
    int y = --x;  // x se decrementa primero, luego se asigna a y
    printf("Decremento prefijo: x = %d, y = %d\n", x, y);
    // Decremento sufijo:
    int z = x--;  // x se asigna a z primero, luego se decrementa
    printf("Decremento sufijo: x = %d, z = %d\n", x, z);
    return 0;
}
```

3.3. Operadores relacionales

Las expresiones relacionales se utilizan para comparar valores. Utilizan operadores como == (igual que), `!=` (no igual que), < (menor que), > (mayor que), <= (menor o igual que) y >= (mayor o igual que). El resultado de estas expresiones es siempre un valor booleano: verdadero o falso. A continuación, se muestra un ejemplo de una expresión con un operador relacional que retorna `0` (falso), ya que no se cumple la condición.

```c
#include <stdio.h>

int main() {
    int a = 5, b = 10;
    int isGreater = (a > b);  // Devuelve 0 (falso)
    return 0;
}
```

3.4. Operadores lógicos

Las expresiones lógicas utilizan operadores lógicos como ***&&*** (AND), **||** (OR) y ***!*** (NOT) para combinar o modificar valores booleanos. A continuación se muestra un ejemplo de una

expresión con un operador lógico que retorna `0` (falso), ya que una de las condiciones no se cumple:

```c
#include <stdio.h>

int main() {
    int x = 5, y = 10;
    int result = (x < y) && (x != 0);  // Devuelve 0 (falso)
    return 0;
}
```

3.5. Operadores de asignación

Las expresiones de asignación, como hemos visto antes, se utilizan para asignar el resultado de una expresión a una variable. El operador de asignación principal es **=**, pero existen operadores de asignación compuesta como `+=`, `-=`, `*=`, entre otros. A continuación, se muestra un ejemplo de un operador de asignación con suma:

```c
#include <stdio.h>

int main() {
    int a = 10;
    a += 5;  // Equivalente a a=a+5; El valor de a será 15
    return 0;
}
```

3.6. Operador condicional (ternario)

Estas expresiones utilizan el operador ternario `? :`, que evalúa una condición y selecciona uno de dos valores posibles. La sintaxis de este operador es `(condición) ? opción1 : opción2;`, y permite seleccionar entre dos opciones en función de si se cumple la condición (`opcion1`) o no (`opción2`). A continuación, se incluye un ejemplo de su uso:

```c
#include <stdio.h>
int main() {
    int a = 5;
    int b = 10;
    int max = (a>b)?a:b; /* Si se cumple la condición (a>b), max
será a; si no, será b. */
    return 0;
}
```

3.7. Operadores de tipo cast

En C, es posible convertir explícitamente un valor de un tipo de dato a otro utilizando una expresión de tipo cast. Para ello se incluye entre paréntesis delante de la variable o valor en cuestión el tipo al que queremos transformar el valor. A continuación, se muestra un ejemplo de conversión de una variable de tipo `int` a un valor de tipo `float`:

```c
#include <stdio.h>
int main() {
    int x = 10;
    float y = (float)x/3; /* Se convierte x en un valor float
para realizar la división */

    return 0;
}
```

3.8. Operadores bit a bit

Los operadores bit a bit permiten manipular datos directamente en su representación binaria, trabajando sobre los bits individuales de los operandos. Cabe destacar que estos operadores también pueden combinarse con el operador de asignación para realizar la operación y asignar el resultado en la misma expresión. A continuación, se presentan los operadores bit a bit disponibles en C.

3.8.1. Desplazamiento a la izquierda (<<)

Desplaza los bits de la variable hacia la izquierda un número de posiciones especificado por el valor de la expresión. El siguiente ejemplo desplaza los bits de la variable `f` una posición hacia la izquierda:

```
int f = 2;  // 2 en binario es 00000010
f = f << 1; // Ahora f vale 4 (00000100)
```

3.8.2. Desplazamiento a la derecha (>>)

Desplaza los bits de la variable hacia la derecha un número de posiciones especificado por el valor de la expresión. El siguiente ejemplo desplaza los bits de la variable `g` dos posiciones hacia la derecha:

```
int g = 8;  // 8 en binario es 00001000
g = g >> 2; // Ahora el valor de g es 2 (00000010)
```

3.8.3. AND bit a bit (&)

Realiza la operación AND bit a bit entre los dos operandos. El siguiente ejemplo realiza una operación AND entre el valor de la variable `h` y `6`, almacenando el resultado en la variable `h`. El resultado de la operación AND tendrá a 1 únicamente los bits que estuvieran a 1 en ambos términos:

```
int h = 6;  // 6 en binario es 00000110
h = h & 3;  // Ahora el valor de h es 2 (00000010)
```

3.8.4. OR bit a bit (|)

Realiza la operación OR bit a bit entre los dos operandos. El siguiente ejemplo realiza una operación OR entre el valor de la variable `i` y `3`, almacenando el resultado de nuevo en la variable `i`. El resultado de la operación OR tendrá a 1 los bits de las posiciones que estuviesen a 1 en al menos uno de los dos términos:

```
int i = 5;  // 5 en binario es 00000101
i = i | 3;  // El valor de i es ahora 7 (00000111)
```

3.8.5. XOR bit a bit (^)

Realiza la operación XOR bit a bit entre los dos operandos. El siguiente ejemplo realiza una operación XOR entre el valor de la variable `j` y `3`, almacenando el resultado de nuevo en la variable `j`. El resultado de la operación XOR tendrá a 1 los bits de las posiciones que estuviesen a 1 únicamente en uno de los dos términos:

```
int j = 5;  // 5 en binario es 00000101
j = j ^ 3;  // El nuevo valor de j es 6 (00000110)
```

3.9. Ejemplo completo

El siguiente ejemplo muestra un programa en C que pide al usuario dos números y luego muestra los resultados de varias expresiones aritméticas, relacionales y lógicas entre esos números:

```c
#include <stdio.h>

int main() {
    int a, b;
    printf("Introduce el primer número:\n");
    scanf("%d", &a);
    printf("Introduce el segundo número:\n");
    scanf("%d", &b);

    // Expresiones aritméticas:
    printf("Suma: %d\n", a+b);
    printf("Multiplicación: %d\n", a*b);

    // Expresiones relacionales:
    printf("Es \'a\' mayor que \'b\': %d\n", a>b);

    // Expresiones lógicas:
    printf("Es \'a\' menor que \'b\' y distinto de cero: %d\n",
(a<b)&&(a!=0));

    return 0;
}
```

En resumen, las expresiones y operadores en C son esenciales para la programación efectiva en cualquier entorno.

4. Conversiones entre tipos

Las conversiones entre tipos en el lenguaje C, son situaciones donde un valor de un tipo de dato se convierte a otro tipo. En C, existen dos tipos principales de conversiones: implícitas y explícitas (también conocidas como casting, las cuales ya se han introducido en el punto 3.7. Operadores de tipo cast). Ambas conversiones son esenciales para evitar errores y comportamientos no deseados en un programa.

4.1. Conversiones implícitas

Es una transformación automática que realiza el compilador de C cuando los operandos de una expresión son de diferentes tipos. Estas conversiones siguen las reglas de promoción de tipos de C, donde se da prioridad a ciertos tipos sobre otros para garantizar que el valor resultante tenga mayor precisión y exactitud.

4.1.1. Reglas de prioridades de los tipos en C

El compilador sigue las siguientes prioridades al realizar conversiones implícitas:

1. `long double`
2. `double`
3. `float`
4. `unsigned long long int`
5. `long long int`

6. `unsigned long int`
7. `long int`
8. `unsigned int`
9. `int`

A continuación, se muestra un ejemplo de conversión implícita:

```
int a = 10;
float b = 5.5;
float result = a + b; /* a se convierte implícitamente a float
antes de realizar la suma */
```

En este caso, la variable `a` es un entero, pero como se está sumando con una variable de tipo `float`, se realiza una conversión implícita del entero a flotante para que la operación se ejecute correctamente.

4.1.2. Promoción de tipos en expresiones

C también realiza promoción de tipos en expresiones aritméticas, principalmente en las operaciones entre enteros de menor tamaño (como `char` y `short`), que son promovidos a `int` automáticamente:

```
char c = 100;
int i = 50;
int result = c + i; /* c se convierte automáticamente a int antes
de realizar la suma */
```

En este caso, `c` (de tipo `char`) se convierte a `int` antes de sumar con `i`.

4.2. Conversiones explícitas (casting)

La conversión explícita, también conocida como casting, se utiliza cuando se desea forzar la conversión de un tipo de dato a otro. En este caso, el programador indica al compilador qué conversión realizar, utilizando la sintaxis `(tipo)` antes de la variable o expresión que se desea convertir:

```
(tipo) expresión
```

A continuación, se incluye un ejemplo de una conversión explícita:

```
int a = 10;
int b = 3;
float result = (float)a/b; /* a se convierte explícitamente a
float antes de la división */
```

En este ejemplo, el casting convierte `a` en un `float` para evitar una división entera y obtener un resultado decimal (3.33333).

4.2.1. Aplicaciones del casting

El casting es especialmente útil cuando se desea realizar operaciones entre tipos de datos diferentes o cuando se necesita ajustar el tipo de un valor a uno más apropiado en un contexto específico, por ejemplo:

```
int num = 260;
char c = (char) num; /* Se convierte num a tipo char, perdiendo
información */
```

Aquí, convertir un entero grande a un `char` puede provocar pérdida de datos, ya que `char` sólo puede almacenar valores pequeños (generalmente entre -128 y 127).

4.1.2. Riesgos del casting

El casting debe usarse con precaución, ya que puede conducir a truncamientos (pérdida de datos) o desbordamientos (si se convierte un tipo más grande a uno más pequeño). Por ejemplo, convertir un `double` a un `int` elimina la parte decimal:

```
double d = 3.14159;
int i = (int) d; // i será igual a 3, se pierde la parte decimal
```

4.1.3. Ejemplo práctico

El siguiente ejemplo corresponde a un programa en C que muestra tanto las conversiones implícitas como explícitas:

```
#include <stdio.h>

int main() {
    // Conversión implícita:
    int x = 10;
    float y = 2.5;
    float sum = x+y; // x se convierte implícitamente a float
    printf("Resultado de la suma: %.2f\n", sum);

    // Conversión explícita (casting):
    int a = 5;
    int b = 2;
    float div = (float) a/b; /* Conversión explícita para obtener
resultado decimal */
    printf("Resultado de la división: %.2f\n", div);

    return 0;
}
```

5. Precedencia y asociatividad

Cuando se escriben expresiones en el lenguaje C que contienen múltiples operadores, el orden en el cual se evalúan dichos operadores se rige por las reglas de precedencia y de asociatividad. Entender estas reglas es fundamental para evitar errores de interpretación y asegurar que las operaciones se ejecuten como se pretende.

5.1. Precedencia

La precedencia de un operador define el orden en el que se evalúan los operadores en una expresión si no hay paréntesis que lo alteren. Los operadores con mayor precedencia se evalúan antes que aquellos con menor precedencia. En el apartado 5.3. Resumen de reglas de precedencia y asociatividad se muestra una tabla con la preferencia de los operadores vistos en este tema. A continuación, se presentan dos ejemplos de cómo actúa la precedencia:

5.1.1. Ejemplo 1

¿Cuál es el valor de la variable `a` en el siguiente código?

```
int a = 2 + 3 * 4;
```

En este caso, la multiplicación tiene mayor precedencia que la suma, por lo que la expresión se evalúa como `a = 2 + (3 * 4)`, resultando en `a = 14`.

5.1.2. Ejemplo 2

¿Cuál será el valor de a en este otro caso?

```
int a = (2 + 3) * 4;
```

En este caso, se fuerza a que la suma se ejecute primero, y luego se realiza la multiplicación, por lo que en este caso `a = 20`.

5.2. Asociatividad

Cuando varios operadores tienen la misma precedencia, el orden en el que se evalúan está determinado por su asociatividad. La asociatividad puede ser de izquierda a derecha o de derecha a izquierda.

5.2.1. Asociatividad de izquierda a derecha

La mayoría de los operadores, entre ellos los operadores aritméticos, los relacionales y los lógicos se evalúan de izquierda a derecha. Por ejemplo:

```
int a = 10 - 3 - 2; // Se evalúa como (10-3)-2, resultado en 5
```

En este caso, los operadores de resta tienen la misma precedencia, y su asociatividad es de izquierda a derecha, por lo que la expresión se evalúa de esa manera. A continuación se incluye otro ejemplo:

```
int x = 2, y = 3, z = 5;
int result = x + y * z - (x + z);
```

En este caso, la expresión se evalúa siguiendo los siguientes pasos para obtener el resultado:

1. Se evalúa la multiplicación `y * z`, ya que tiene mayor precedencia: `3 * 5 = 15`.
2. Se evalúa la suma dentro de los paréntesis: `x + z = 2 + 5 = 7`.
3. Se evalúan las sumas y restas restantes: `x + 15 - 7 = 2 + 15 - 7 = 10`.

Por lo tanto, el valor final de *`result`* es `10`.

5.2.2. Asociatividad de derecha a izquierda

Algunos operadores se evalúan de derecha a izquierda. En este grupo encontramos los operadores de asignación, el operador ternario, los operadores de dirección e indirección, los de pre-incremento y pre-decremento y los castings. A continuación, se muestra un ejemplo con operadores de asignación:

```
int a, b, c;
a = b = c = 5;
```

Esta expresión se evalúa de derecha a izquierda: primero, `c` recibe el valor `5`, luego `b` recibe el valor de `c` (que ahora es `5`), y finalmente, `a` recibe el valor de `b` (también `5`). Se incluye también un ejemplo con un operador condicional ternario:

```
int x = 10, y = 20, z;
z = (x > y) ? x : y;
```

El operador ternario se evalúa también de derecha a izquierda, comenzando con la comparación `(x > y)`, y luego seleccionando el valor adecuado dependiendo de si la condición es verdadera (primera expresión, en este caso el valor de `x`) o falsa (segunda expresión, en este caso el valor de `y`).

5.3. Resumen de reglas de precedencia y asociatividad

La siguiente tabla resume las reglas de precedencia y asociatividad de los operadores de C vistos en este tema:

Precedencia	Operadores	Tipo de operación	Asociatividad
1	*() [] -> .*	Acceso a miembros, llamada de función	Izquierda a derecha
2	*! ~ ++ -- + - * & (tipo)*	Operadores unarios y cast	Derecha a izquierda
3	** / %*	Multiplicación, división, módulo	Izquierda a derecha
4	*+ -*	Suma, resta	Izquierda a derecha
5	*<< >>*	Desplazamientos	Izquierda a derecha
6	*< <= > >=*	Comparaciones	Izquierda a derecha
7	*== !=*	Igualdad, desigualdad	Izquierda a derecha
8	*&*	AND bit a bit	Izquierda a derecha
9	*^*	XOR bit a bit	Izquierda a derecha
10	*|*	OR bit a bit	Izquierda a derecha
11	*&&*	AND lógico	Izquierda a derecha
12	*||*	OR lógico	Izquierda a derecha
13	*? :*	Operador ternario	Derecha a izquierda
14	*= += -= *= /= %= <<= >>= &= ^=*	Asignación	Derecha a izquierda
15	*,*	Evaluar múltiples expresiones	Izquierda a derecha

6. Biblioteca math.h en C

En programación, una biblioteca es un conjunto de funciones predefinidas que un programador puede incluir en sus proyectos para realizar tareas específicas sin tener que implementarlas desde cero. Las bibliotecas permiten ahorrar tiempo y esfuerzo, proporcionando soluciones optimizadas para tareas comunes como operaciones matemáticas, manipulación de cadenas, o manejo de archivos.

En C, las bibliotecas se incluyen mediante la directiva `#include`, que permite al compilador acceder a las funciones de dicha biblioteca. Un ejemplo típico es la biblioteca de entrada/salida estándar `stdio.h`, que proporciona funciones para la entrada y salida, como `printf` y `scanf`.

Por su parte, la biblioteca `math.h` en C es una colección de funciones matemáticas que permiten realizar operaciones complejas como potencias, raíces, funciones

trigonométricas, logaritmos, etc. Esta biblioteca es muy útil cuando se trabaja con cálculos numéricos avanzados.

Para utilizar las funciones de la biblioteca `math.h`, se debe incluir en el archivo fuente con la directiva:

```
#include <math.h>
```

A continuación, se muestran algunas de las funciones más relevantes de esta biblioteca:

Función	Uso	Ejemplo de Utilización
sqrt(x)	Devuelve la raíz cuadrada de x.	double r = sqrt(25.0);
pow(x, y)	Calcula x elevado a la potencia y.	double p = pow(2.0, 3.0);
sin(x)	Devuelve el seno de x en radianes.	double s = sin(M_PI/2);
cos(x)	Devuelve el coseno de x en radianes.	double c = cos(0);
tan(x)	Devuelve la tangente de x en radianes.	double t = tan(M_PI/4);
log(x)	Devuelve el logaritmo natural de x (base e).	double l = log(2.718);
log10(x)	Devuelve el logaritmo en base 10 de x.	double l10 = log10(100);
exp(x)	Calcula e elevado a la potencia x.	double e = exp(1);
fabs(x)	Devuelve el valor absoluto de x.	double abs_val = fabs(-5.0);
ceil(x)	Redondea x al siguiente número entero mayor o igual.	double up = ceil(2.3);
floor(x)	Redondea x al siguiente número entero menor o igual.	double down = floor(2.7);
fmod(x, y)	Devuelve el resto de x / y (modulo).	double mod = fmod(7.5, 2.3);
round(x)	Redondea x al entero más cercano.	double r = round(2.6);

6.1. Ejemplos de uso de math.h en C

En este apartado se incluyen diferentes ejemplos de código en C que hace uso de funciones de la biblioteca math.h:

6.1.1. Cálculo de una raíz cuadrada

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

int main() {
    double num = 16.0;
    double result = sqrt(num);
    printf("La raíz cuadrada de %.2f es %.2f\n", num, result);
    return 0;
}
```

En este ejemplo, `sqrt` calcula la raíz cuadrada del número `16.0`, y la salida del programa será:

```
La raíz cuadrada de 16.00 es 4.00
```

6.1.2. Potencia de un número

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

int main() {
    double base = 2.0, exp = 3.0;
    double result = pow(base, exp);
    printf("%.2f elevado a %.2f es %.2f\n", base, exp, result);
    return 0;
}
```

En este otro ejemplo, `pow` calcula `2` elevado a `3`, obteniendo `8`. La salida de este programa será:

```
2.00 elevado a 3.00 es 8.00
```

6.1.3. Cálculo de funciones trigonométricas

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

int main() {
    double angulo = M_PI / 4; // 45 grados en radianes
    printf("Seno: %.2f\n", sin(angulo));
    printf("Coseno: %.2f\n", cos(angulo));
    printf("Tangente: %.2f\n", tan(angulo));
    return 0;
}
```

En este ejemplo, `sqrt` calcula la raíz cuadrada del número `16.0`, y la salida del programa será:

```
Seno: 0.71
Coseno: 0.71
Tangente: 1.00
```

En este último ejemplo, las funciones trigonométricas `sin`, `cos` y `tan` calculan los valores del seno, coseno y tangente de `45` grados (expresados en radianes).

Tema 4: Estructuras de Control

Las instrucciones vistas hasta ahora se han ejecutado de manera secuencial, siguiendo el orden en el que se encuentran escritas en nuestro código. Sin embargo, en ocasiones es muy útil modificar ese orden y controlar el flujo de ejecución. En este contexto destacan las estructuras de control, que permiten definir qué instrucciones se ejecutan en cada momento. Existen dos tipos de estructuras de control, las cuales se verán en detalle en este tema.

- Estructuras de selección: permiten seleccionar si un conjunto de instrucciones debe o no ejecutarse.
- Estructuras de repetición: permiten repetir la ejecución de una o varias instrucciones.

1. Estructuras de selección

Las estructuras de selección permiten variar el flujo de ejecución de un programa dependiendo de si se cumplen o no ciertas condiciones. En C, las principales estructuras de selección son `if`, `if-else`, `switch-case` y el operador ternario `?:`. Estas estructuras nos permitirán diseñar programas dinámicos que permitan ejecutar diferentes bloques de código en función de las condiciones especificadas.

1.1. Estructura `if`

La estructura `if` permite ejecutar una instrucción o bloque de instrucciones únicamente si la condición especificada se evalúa como verdadera (o cualquier número diferente de 0 en C). Usualmente esta condición contendrá operadores relacionales y lógicos, entre otros posibles.

1.1.1. Sintaxis

La sintaxis de la estructura `if` es la siguiente:

```
if (condición)
{
    // Instrucciones a ejecutar
}
```

Nótese que esta estructura no debe llevar `;`, ya que de incluirlo se entenderá que la estructura no incluye ningún bloque de instrucciones. Las llaves `{ }` indican que, en caso de que la condición se evalúe como verdadera, debe ejecutarse todo el bloque de instrucciones que se encuentran entre las llaves. En caso de no incluirlas, se entiende que únicamente debe ejecutarse la instrucción inmediatamente posterior a la estructura `if`.

1.1.2. Ejemplos

A continuación, se muestra un ejemplo de un programa que muestra el mensaje "`Eres mayor de edad`" en caso de que la variable `edad` sea igual o mayor que `18`. En caso de que la variable `edad` fuese menor que `18`, el programa no mostraría nada:

©2024 Diego Hortelano Haro, Gerardo Reyes Salgado, Manuel Rubio Sánchez. Algunos derechos reservados. Este documento se distribuye bajo la licencia "Atribución/Reconocimiento-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en https://creativecommons.org/licenses/by-sa/4.0/deed.es.

```c
#include <stdio.h>

int main() {
    int edad = 20;

    if (edad >= 18) {
        printf("Eres mayor de edad.\n");
    }

    return 0;
}
```

El siguiente ejemplo muestra por pantalla los mensajes "`Hace calor`" y "`Es un buen día para ir a la playa`" si el número leído por teclado es mayor que `0`.

```c
#include <stdio.h>

int main() {
    int temperatura;

    printf("Dame la temperatura actual\n");
    scanf("%d", &temperatura);

    if (temperatura > 25) {
        printf("Hace calor.\n");
        printf("Es un buen día para ir a la playa.\n");
    }

    return 0;
}
```

Nótese que en el primer ejemplo es posible omitir las llaves, ya que el bloque de instrucciones contiene una única instrucción. Sin embargo, esto no es posible en el segundo ejemplo sin alterar el funcionamiento de este.

1.2. Estructura `if-else`

La estructura `if-else` permite ejecutar una o varias instrucciones si una condición es verdadera y otra u otras instrucciones si dicha condición es falsa.

1.2.1. Sintaxis

La sintaxis de la estructura `if-else` es la siguiente:

```c
if (condición)
{
    // Instrucciones a ejecutar si la condición es verdadera
} else {
    // Instrucciones a ejecutar si la condición es falsa
}
```

Al igual que la estructura `if`, `if-else` tampoco debe llevar `;`, ya que de incluirlo el programa entenderá que la estructura no incluye ningún bloque de instrucciones. De nuevo aquí, las llaves `{ }` permiten incluir múltiples instrucciones que serán ejecutadas en función de la condición dada.

1.2.2. Ejemplos

El siguiente ejemplo muestra un programa que lee la edad del usuario e imprime "`Eres mayor de edad`" en caso de que la variable `edad` sea igual o mayor que `18` o "`Eres menor de edad`" en caso contrario:

```c
#include <stdio.h>

int main() {
    int edad;

    printf("Inserta tu edad\n");
    scanf("%d", &edad);

    if (edad >= 18) {
        printf("Eres mayor de edad.\n");
    } else {
        printf("Eres menor de edad.\n");
    }

    return 0;
}
```

El siguiente ejemplo muestra dos estructuras `if-else` encadenadas. En este caso la evalúa la condición de la estructura superior, ejecutándose el bloque de instrucciones que imprime "`El número es positivo`". En caso contrario se pasa a la cláusula `else`, la incluye una estructura `if-else` completa. Esta segunda estructura solamente se evaluará en caso de la condición de la cláusula `if` superior haya sido falsa, y evaluará una nueva condición. En caso de que esta condición se evalúe como verdadera se ejecutará el bloque de instrucciones que incluye "`El número es negativo`". Finalmente, si ninguna de las condiciones se evalúa como verdadera, se ejecutará el último bloque de instrucciones, que imprimirá "`El número es cero`".

```c
#include <stdio.h>

int main() {
    int numero;

    scanf("%d", &numero);

    if (numero > 0) {
        printf("El número es positivo.\n");
        printf("Se mostrará si la primera condición es
verdadera.\n");
    } else if (numero < 0) {
        printf("El número es negativo.\n");
        printf("Se mostrará si la primera condición es falsa y la
segunda verdadera.\n");
    } else {
        printf("El número es cero.\n");
        printf("Se mostrará si ambas condiciones son falsas.\n");
    }

    return 0;
}
```

Al igual que sucede con la estructura `if`, en la estructura `if-else` pueden omitirse las llaves si deseamos ejecutar una única instrucción.

1.3. switch-case

La estructura `switch-case` permite seleccionar entre múltiples bloques de instrucciones a ejecutar, basándose en el valor obtenido al resolver una expresión.

1.3.1. Sintaxis

A continuación, se muestra la sintaxis de la estructura `switch-case`. Como puede observarse, al inicio de esta estructura se incluye la expresión que se evaluará y permitirá seleccionar qué bloque se ejecutará. Esta expresión puede incluir diferentes operadores, pero su valor final debe ser un valor entero.

```
switch (expresión) {
    case valor1:
        // Código a ejecutar si expresión == valor1
        break;
    case valor2:
        // Código a ejecutar si expresión == valor2
        break;
    // Más casos...
    default:
        // Código a ejecutar si ningún valor coincide
}
```

Como puede verse, esta estructura cuenta con múltiples cláusulas `case` (podemos incluir tantas como necesitemos), las cuales incluyen el valor entero que debe coincidir con el obtenido por la expresión para ejecutar el bloque en cuestión. Cada cláusula `case` suele finalizar con una instrucción `break`, la cual indica a nuestro programa hasta qué instrucción debe ejecutar. En caso de no incluir la sentencia `break` en alguna cláusula `case`, nuestro programa continuará ejecutando instrucciones, pasando al siguiente `case` hasta llegar a una sentencia `break` o hasta finalizar la estructura `swith-case`. Finalmente, la cláusula `default` es opcional, y permite incluir un bloque de instrucciones que se ejecutará si el valor de la expresión no coincide con ninguno de los indicados en las diferentes cláusulas `case`.

También es posible incluir un rango de valores en una cláusula `case` (en lugar de un único valor entero), indicando los dos extremos del intervalo. Para ello, basta con indicar en las cláusulas `case` los extremos de los intervalos, separados con tres puntos, tal y como se muestra a continuación:

```
switch (expresión) {
    case inicio_intervalo1 ... fin_intervalo1:
        // Código a ejecutar si expresión == valor1
        break;
    case inicio_intervalo2 ... fin_intervalo2:
        // Código a ejecutar si expresión == valor2
        break;
    // Más casos...
    default:
        // Código a ejecutar si ningún valor coincide
}
```

1.3.2. Ejemplos

A continuación, se muestra un ejemplo donde se pide al usuario que inserte un número del 1 al 7 (ambos incluidos). Tras leer dicho número, se utiliza una estructura `switch-case` para mostrar por pantalla el día correspondiente a dicho número. En caso de que el usuario introduzca un número con un valor fuera de dicho intervalo, se mostrará el mensaje indicado en la cláusula `default`:

```c
#include <stdio.h>

int main() {
    int dia;

    printf("Inserta número del 1 al 7 para indicar un día:\n");
    scanf("%d", &dia);

    switch (dia) {
        case 1:
            printf("Lunes\n");
            break;
        case 2:
            printf("Martes\n");
            break;
        case 3:
            printf("Miércoles\n");
            break;
        case 4:
            printf("Jueves\n");
            break;
        case 5:
            printf("Viernes\n");
            break;
        case 6:
            printf("Sábado\n");
            break;
        case 7:
            printf("Domingo\n");
            break;
        default:
            printf("Día no válido\n");
    }

    return 0;
}
```

El siguiente ejemplo muestra una estructura `switch-case` que ejecutará bloques de instrucciones con múltiples instrucciones. Una vez que se selecciona un bloque de instrucciones, se continuará con la ejecución hasta el primer `break`:

```c
#include <stdio.h>

int main() {
    int opcion;
    printf("Elige una opción (1-2): ");
    scanf("%d", &opcion);
```

```
    switch(opcion) {
        case 1:
            printf("Has elegido la opción 1.\n");
            printf("Realizando acción 1...\n");
            // Más instrucciones para la opción 1
            break;
        case 2:
            printf("Has elegido la opción 2.\n");
            printf("Realizando acción 2...\n");
            // Más instrucciones para la opción 2
            break;
        default:
            printf("Opción no válida.\n");
    }

    return 0;
}
```

En caso de no encontrarse una sentencia `break`, la ejecución continuará de manera secuencial, ejecutando instrucciones de otras cláusulas `case`. En el siguiente ejemplo, si el usuario inserta una vocal, al no haber sentencias `break` en las primeras cláusulas `case`, la ejecución continuará hasta que se muestre el mensaje `"Has introducido una vocal"`. Además, recordemos que el tipo `char` se almacena internamente como un valor entero, por lo que también es posible utilizarlos en una estructura `switch-case`.

```
#include <stdio.h>

int main() {
    char letra;
    printf("Introduce una letra:\n");
    scanf("%c", &letra);

    switch(letra) {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
            printf("Has introducido una vocal.\n");
            // Instrucciones comunes para las vocales
            break;
        default:
            printf("No has introducido una vocal.\n");
    }

    return 0;
}
```

Finalmente, el último ejemplo muestra cláusulas `case` con intervalos, siendo el primero de 0 a 17 (ambos incluidos) y el segundo de 18 a 120 (también ambos incluidos). En este caso, se ha omitido la cláusula `default` ya que su uso no es obligatorio.

```
#include <stdio.h>

int main() {
    int edad;
    printf("Introduce tu edad:\n");
```

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

    switch(edad) {
        case 0 ... 17:
            printf("Aún eres menor de edad.\n");
            break;
        case 18 ... 120:
            printf("Ya eres mayor de edad.\n");
            break;
    }

    return 0;
}
```

1.4. Operador Ternario ? :

El operador ternario `? :` condicional permite escribir de forma compacta una expresión que asigna un valor basándose en la evaluación de una condición.

1.4.1. Sintaxis

La sintaxis de este operador incluye la condición a evaluar y dos expresiones. En caso de que la condición se evalúe como verdadera se devolverá la primera expresión y, en caso contrario, la segunda:

```
condición ? expresión_si_verdadera : expresión_si_falsa;
```

1.4.2. Ejemplos

A continuación, se incluye un ejemplo que usa este operador ternario condicional para seleccionar el menor número entre dos números leídos por teclado. En caso de que la condición (`num1 < num2`) resulte verdadera, la variable `minimo` recibirá el valor de `num1`, mientras que si la condición se evalúa como falsa, la variable `minimo` recibirá el valor de `num2`.

```
#include <stdio.h>

int main() {
    int minimo;
    int num1, num2;

    printf("Inserta dos números:\n");
    scanf("%d %d", &num1, &num2);

    minimo = (num1 < num2) ? num1 : num2;
    printf("El mínimo es: %d\n", minimo);

    return 0;
}
```

También es posible asignar una cadena de caracteres. El siguiente ejemplo asigna al código de control relativo a las cadenas de caracteres (`"%s"`) la cadena `"Sí"` o la cadena `"No"` en función de la condición evaluada.

```
#include <stdio.h>

int main() {
    int edad;
```

```
    printf("Inserta tu edad:\n");
    scanf("%d", &edad);

    printf("¿Eres mayor de edad? %s\n", (edad >= 18) ? "Sí" : "No");

    return 0;
}
```

De igual manera, el operador ternario condicional también permite ejecutar instrucciones, tal y como muestra el siguiente ejemplo. Sin embargo, en estos casos es preferible utilizar una estructura `if-else` para mejorar la legibilidad, reservando el operador ternario para retornar valores simples.

```
#include <stdio.h>

int main() {
    int edad;

    printf("Inserta tu edad:\n");
    scanf("%d", &edad);

    (edad >= 18) ? printf("Eres mayor de edad.\n") : printf("Aún no
eres mayor de edad");

    return 0;
}
```

2. Estructuras de repetición

Las estructuras de repetición también permiten variar el flujo de ejecución secuencial de un programa, pero, en este caso, repiten una o varias instrucciones. Las estructuras de repetición disponibles en el lenguaje C son `for`, `while` y `do-while`. Estas estructuras permiten ejecutar un bloque de código múltiples veces, lo que resulta fundamental a la hora de automatizar tareas repetitivas, trabajar con colecciones de datos o controlar eficientemente el flujo de nuestro programa.

2.1. Estructura for

El bucle `for` permite repetir una instrucción o un bloque de instrucciones mientras que se cumpla la condición indicada. A diferencia del bucle `while`, el bucle `for` se utiliza cuando se conoce previamente el número de iteraciones que se debe realizar.

2.1.1. Sintaxis

La estructura `for` incluye unos paréntesis "`()`" con tres campos, separados por "`;`", y **no** incluye un "`;`" final. Los campos de la estructura `for` son:

- Inicialización: permite inicializar variables que se utilicen en el bloque de instrucciones. Usualmente se utiliza para inicializar la variable contador que varía en cada iteración.
- Condición: permite seleccionar si el bloque de instrucciones debe ejecutarse o no. Dicho bloque se ejecutará mientras que la condición incluida aquí se evalúe como verdadera. Usualmente esta condición incluirá operadores relacionales y lógicos, entre otros, así como a la variable inicializada en el campo anterior. Esta condición

deberá evaluarse como falsa en algún momento para que el bloque de instrucciones deje de repetirse y nuestro programa pueda continuar.

- Actualización: permite actualizar las variables utilizadas en el bloque, aunque usualmente se utiliza para actualizar el valor de la variable contador.

La sintaxis del bucle `for` se muestra a continuación:

```
for (inicialización; condición; actualización) {
    // Código a ejecutar
}
```

A la hora de ejecutar una estructura `for` se siguen los siguientes pasos:

1. Se ejecuta la sentencia de `inicialización`.
2. Se evalúa la `condición`.
    a. Si la condición es cierta, se ejecuta todo el bloque de instrucciones (encerrado entre `{}`). Cuando finaliza la iteración, se ejecuta la sentencia de `actualización` y se vuelve a evaluar la `condición` (paso 2).
    b. Si la condición es falsa, el bucle finaliza.

Así, es de vital importancia elegir correctamente las sentencias de `inicialización`, `condición` y `actualización` de una estructura `for`. La `condición` debe evaluarse como verdadera al comienzo (en caso contrario, el bucle `for` nunca ejecutará el bloque de instrucciones), y debe poder pasar a evaluarse como falsa en algún momento (ya que de lo contrario nunca dejaremos de repetir el bloque de instrucciones, entrando en un bucle infinito).

2.1.2. Ejemplos

La estructura `for` del siguiente ejemplo inicializa una variable `i` a `1`, incrementándola en uno al final de cada iteración. Como la condición establece que el valor de `i` debe ser menor o igual a `10`, el bloque de código se repetirá un total de 10 veces, una para cada valor diferente de `i`. Así, el programa mostrará 10 líneas, mostrando los números del 1 al 10 (los valores de `i` en cada iteración).

```
#include <stdio.h>

int main() {
    for (int i = 1; i <= 10; i++) {
        printf("%d\n", i);
    }
    return 0;
}
```

El siguiente ejemplo presenta una estructura `for` que inicializa la variable `i` a `1` y la incrementa en una unidad en cada iteración. Como la condición establece que `i` debe ser menor o igual a `5`, el bucle realizará 5 repeticiones, variando el valor de `i`. Por otra parte, la variable `suma` va acumulando la suma de los diferentes valores de `i`. Este tipo de variables se denominan acumuladores, para lo que deben estar en ambos lados de la asignación o hacer uso de los operadores combinados con el operador de asignación (=), como en este caso. En caso contrario, el valor almacenado en una iteración sobrescribirá al anterior. Destaca también aquí el valor inicial de esta variable `suma`, ya que debemos seleccionar

un valor que no interfiera con el resultado final (en este caso, el elemento neutro de la suma, un `0`). Finalmente, el programa imprime el valor de `suma`, la suma de los 5 primeros números naturales.

```c
#include <stdio.h>

int main() {
    int suma = 0;
    for (int i = 1; i <= 5; i++)
        suma += i;

    printf("La suma es: %d\n", suma);
    return 0;
}
```

También es posible utilizar el operador de coma (`,`) para incluir múltiples expresiones en las sentencias de `inicialización` y `actualización`. El siguiente ejemplo hace uso de este operador para incrementar e imprimir la variable `i` a la vez que decrementa e imprime la variable `j`:

```c
#include <stdio.h>

int main() {
    for (int i = 0, j = 10; i < j; i++, j--) {
        printf("i: %d, j: %d\n", i, j);
    }
    return 0;
}
```

2.2. Estructura while

El bucle `while` también permite repetir una instrucción o bloque de instrucciones mientras se cumpla la condición incluida. En este caso, a diferencia del bucle `for`, el bucle `while` suele utilizarse cuando no se conoce exactamente el número de iteraciones (por ejemplo, porque dependa de valores insertados por el usuario o porque trabajemos con listas dinámicas que puedan variar su tamaño).

2.2.1. Sintaxis

La estructura `while` incluye la condición a evaluar, que puede combinar diferentes tipos de operadores, y el bloque de código a ejecutar en caso de que la condición se evalúe como verdadera. A continuación, se muestra la sintaxis básica de la estructura `while`.

```c
while (condición) {
    // Código a ejecutar
}
```

A la hora de ejecutar una estructura `while` se siguen los siguientes pasos:

1. Se evalúa la `condición`.
    a. Si la `condición` es cierta, se ejecuta todo el bloque de instrucciones., pasando de nuevo a evaluar la `condición` (paso 1).

b. Si la `condición` es falsa, el bucle finaliza y el bloque de instrucciones no se ejecuta.

En este caso, aunque la estructura `while` no incluya la sentencia de `inicialización`, las variables utilizadas en la condición deben tener un valor previo. Así mismo, dentro del bloque de instrucciones debe ejecutarse alguna instrucción que permita cambiar la evaluación de la condición a falso en alguna iteración (`actualización`), para que el bucle pueda finalizar.

2.2.2. Ejemplos

El siguiente ejemplo muestra el uso de una estructura `while` utilizada para imprimir los 10 primeros números naturales. En este caso, la variable `i` debe incrementarse en el bloque de instrucciones para que la condición pueda pasar a evaluarse como falsa en alguna iteración y el bucle pueda finalizar.

```c
#include <stdio.h>

int main() {
    int i = 1;
    while (i <= 10) {
        printf("%d\n", i);
        i++;
    }
    return 0;
}
```

El siguiente ejemplo muestra el uso de la estructura `while` para sumar los 5 primeros números naturales. Para ello, hace uso de una variable `suma` que acumula en cada iteración el valor de la variable `i` (que va incrementándose desde 1 hasta 5).

```c
#include <stdio.h>

int main() {
    int suma = 0;
    int i = 1;
    while (i <= 5) {
        suma += i;
        i++;
    }
    printf("La suma es: %d\n", suma);
    return 0;
}
```

Como puede observarse, ambos ejemplos son los mismos que los propuestos para la estructura `for`, y es que cualquier bucle puede realizarse utilizando cualquier forma. Por ejemplo, podemos pasar fácilmente de un bucle `for` a un bucle `while` realizando la sentencia de `inicialización` previamente e incluyendo la sentencia de `actualización` en el bloque de instrucciones a repetir.

2.3. do-while

El bucle `do-while` es muy similar al bucle `while` visto anteriormente, incluso su sintaxis es muy parecida, pero hay una diferencia clave, y es que el bucle `do-while` ejecuta el

bloque de instrucciones siempre al menos una vez, ya que comprueba la condición tras la ejecución.

2.3.1. Sintaxis

La estructura del bucle `do-while` comienza por el bloque de código a repetir, situado entre las sentencias `do` y `while` de la estructura. Esta última sentencia incluye la condición a evaluar, que, al igual que sucede en el resto de bucles, puede combinar diferentes tipos de operadores. A continuación, se muestra la sintaxis básica de la estructura `while`.

```
do {
    // Código a ejecutar
} while (condición);
```

A la hora de ejecutar una estructura `do-while` se siguen los siguientes pasos:

1. Se ejecuta el bloque de instrucciones.
2. Se evalúa la `condición`.
    a. Si la `condición` es cierta, se vuelve a ejecutar todo el bloque de instrucciones (paso 1).
    b. Si la `condición` es falsa, el bucle finaliza.

En este caso, también debemos tener en cuenta que las variables utilizadas en la `condición` deben tener un valor válido a la hora de evaluar la condición, y que dicha condición debe evaluarse como falsa en algún momento para permitir salir del bucle y finalizar nuestro programa.

2.3.2. Ejemplos

El siguiente ejemplo utiliza un bucle `do-while` para mostrar por pantalla el valor de la variable `i`, el cual se incrementa en cada iteración, pasando por todos los valores desde `1` hasta `10`.

```
#include <stdio.h>

int main() {
    int i = 1;

    do {
        printf("%d\n", i);
        i++;
    } while (i <= 10);

    return 0;
}
```

El siguiente ejemplo obtiene el resultado de sumar los 5 primeros números enteros, el cual muestra por pantalla una vez finalizado el bucle. Para ello se utiliza una variable (suma) que acumula el valor de cada uno de los números, siendo necesario que dicha variable aparezca en ambos lados del operador de asignación (o se utilice una combinación de un operador aritmético con el operador de asignación, como en este caso):

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

```c
int main() {
    int suma = 0;
    int i = 1;

    do {
        suma += i;
        i++;
    } while (i <= 5);

    printf("La suma es: %d\n", suma);
    return 0;
}
```

El siguiente ejemplo muestra un programa que lee números de teclado hasta que el usuario inserte un `7`. Si bien en los ejemplos anteriores es posible cambiar la estructura `do-while` por una estructura `for` o una estructura `while`, en este caso no es posible hacerlo sin realizar antes una primera lectura (permitiendo así que la variable `num` tenga un valor válido antes de evaluar la condición). Es en estos casos donde el bucle `do-while` cobra especial relevancia.

```c
#include <stdio.h>

int main() {
    int num;

    do{
        scanf("%d", &num);
    }while(num != 7);

    return 0;
}
```

2.4. Sentencias break y continue en estructuras de repetición

A lo largo de este tema ya se ha hablado sobre la sentencia `break`. Tanto esta sentencia como la sentencia `continue` pueden utilizarse con bucles.

Como se ha detallado antes, la sentencia `break` permite detener la ejecución de un bloque de instrucciones, permitiendo finalizarla ejecución de un bucle. Sin embargo, la mayoría de autores **no recomiendan su uso** en bucles ya que suele hacer el código más impredecible y difícil de seguir, pudiendo ocasionar errores si no tenemos cuidado. A continuación, se muestra un ejemplo del uso de la sentencia `break`, la cual finaliza la ejecución del bucle infinito (la `condición` siempre se evalúa como verdadera) cuando `i` tiene un valor de `5`. Como puede observarse, es posible sustituir el uso de la sentencia `break` por una condición equivalente en la sentencia `while`:

```c
int i = 0;
while(1)
{
    printf("%i\n", i);
    i++;

    if(i == 5) {
        break;
    }
}
```

Por otra parte, la sentencia `continue` también finaliza la ejecución del bloque de instrucciones de un bucle, pero en este caso pasa a ejecutar la siguiente iteración del bucle. De nuevo en este caso, **no se recomienda su uso**, ya que hacen nuestro código menos legible y pueden provocar errores si no se utiliza con cuidado. El siguiente ejemplo mostrará el valor de `i` cuando la variable tenga un valor de `0` (primera iteración) y `2` (última iteración), pero con el valor `1` se ejecutará la sentencia `continue`, saltando el resto de las instrucciones del bloque (en este caso, únicamente la instrucción de mostrar la variable).

```
for (int i = 0; i < 3; i++) {
    // Si i es igual a 1, saltar a la siguiente iteración
    if (i == 1) {
        continue;
    }
    printf("El valor de i es: %d\n", i);
}
```

3. Evaluación perezosa

En C, las evaluaciones de las expresiones lógicas (como las utilizadas en las estructuras de selección y de repetición detalladas en este tema) se realizan siguiendo una evaluación perezosa, también conocida como evaluación en cortocircuito.

Al hacer uso de una evaluación perezosa, la evaluación se detiene tan pronto como se conoce el resultado, lo que es de especial interés cuando tenemos varias expresiones unidas con operadores lógicos. Esto permite ahorrar tiempo de ejecución, pero también influye en el orden de las expresiones, las cuales debemos seleccionar con cuidado para optimizar nuestro programa. Por ejemplo, el siguiente código pide al usuario que inserte dos números por teclado y muestra si la división entre ellos es exacta.

```
int a, b;
scanf("%d %d", &a, &b);
if(a % b == 0){
    printf("La división %d/%d es exacta\n", a, b);
}
```

Sin embargo, si el usuario inserta un 0 en el segundo número, nuestro programa fallará, dando un error y no mostrando ninguna salida. Una posible solución sería incluir una nueva estructura `if` para comprobar esto:

```
int a, b;
scanf("%d %d", &a, &b);
if(b != 0) {
    if(a % b == 0) {
        printf("La división %d/%d es exacta\n", a, b);
    }
}
```

Otra posible solución es aprovechar la evaluación perezosa realizada por C para incluir ambas condiciones en el mismo `if`, sabiendo que la división nunca se realizará si la primera condición no se cumple. Esto se debe a que, al estar unidas con un operador lógico `AND`, cada una de las condiciones deben evaluarse como verdaderas para que la expresión se considere verdadera:

```c
int a, b;
scanf("%d %d", &a, &b);
if ((b != 0) && (a % b == 0)) {
    printf("La división %d/%d es exacta\n", a, b);
}
```

Nótese que en este caso no sería posible cambiar el orden de las condiciones, ya que de lo contrario nuestro programa podría intentar realizar una división entre 0, finalizando con un error.

Tema 5: Arrays y Cadenas de Caracteres

Los tipos de datos básicos y derivados de C vistos en el Tema 2 nos permiten almacenar valores de diferentes tipos. Sin embargo, estos tipos únicamente permiten declarar variables con un único valor. En C existen otros tipos de datos, denominados datos compuestos, que permiten almacenar múltiples valores. Estos tipos son:

- Arrays y cadenas de caracteres: permiten almacenar múltiples elementos del mismo tipo de forma contigua en memoria.
- Estructuras: permiten agrupar valores de diferentes tipos en una única variable.

Este tema se centra en los arrays y cadenas de caracteres, los cuales son fundamentales para el manejo eficiente de datos en C. De igual manera, se presenta la biblioteca `string.h`, que será de gran utilidad al trabajar con cadenas de caracteres.

1. Tipos de datos compuestos

Los tipos de datos compuestos permiten agrupar múltiples elementos de datos del mismo tipo o de diferentes tipos en una sola estructura. En C, existen varios tipos de datos compuestos que nos permiten manejar colecciones de datos. Los más comunes son los **arrays**, las **cadenas de caracteres** y las **estructuras**. Centrándonos en los que permiten definir múltiples elementos del mismo tipo podemos distinguir:

- **Arrays (o arreglos)**: son colecciones de elementos del mismo tipo almacenados de forma contigua en memoria. Cada elemento del array debe accederse mediante un índice.
- **Cadenas de caracteres**: son arrays de tipo `char` que permiten almacenar secuencias de caracteres, finalizadas con un carácter nulo (`'\0'`). Las cadenas se usan comúnmente para trabajar con texto.

2. Arrays

Un `array` es una estructura de datos que permite almacenar múltiples elementos del mismo tipo en una única variable, utilizando un espacio contiguo de memoria. Los `arrays` son útiles cuando se necesita almacenar una secuencia de valores relacionados, como una lista de números o una serie de caracteres.

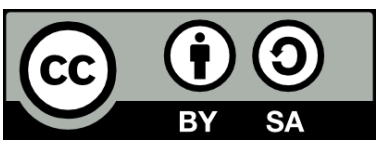
2.1. Declaración

En C, la declaración de un array tiene la siguiente forma general:

```
tipo id_array [num_elementos];
```

Donde:

- `tipo`: es el tipo de datos de los elementos del array (por ejemplo, `int`, `float`, o `char`).
- `id_array`: es el nombre del array. Debemos utilizar un identificador válido según las reglas detalladas en el Tema 2.
- `num_elementos`: es el número de elementos que el array puede contener.



©2024 Diego Hortelano Haro, Gerardo Reyes Salgado, Manuel Rubio Sánchez. Algunos derechos reservados. Este documento se distribuye bajo la licencia "Atribución/Reconocimiento-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en https://creativecommons.org/licenses/by-sa/4.0/deed.es.

2.1.1. Ejemplos

A continuación, se muestra la declaración de 3 arrays de diferentes tipos: en primer lugar, un array de enteros de 5 elementos; en segundo lugar, un array de 10 elementos en punto flotante; finalmente, un array de 3 caracteres:

```
// ejemplos de declaraciones de array
int numeros[5];  // array de 5 enteros
float temperaturas[10];  // array de 10 elementos en punto flotante
char letras[3];  // array de 3 caracteres
```

2.2. Acceso y errores

Los elementos de un array se acceden usando el identificador del array junto al índice de la posición del elemento que queremos acceder. Dicho índice se incluye entre corchetes (`[]`) junto al identificador. En C, los índices de los arrays comienzan en 0, lo que significa que el primer elemento tiene el índice 0, el segundo el índice 1, y así sucesivamente. Por lo tanto, la última posición válida para un array de `n` elementos será el índice `n-1`.

2.2.1. Ejemplos

El siguiente ejemplo muestra el acceso al primer elemento de un array, al cual se accede en primer lugar para asignarle un valor y posteriormente para mostrarlo por pantalla:

```
// ejemplos de acceso de un array
num[0] = 10;  // Asignar 10 al primer elemento del array "num"
printf("%d\n", num[0]);  // Imprimir el valor del primer elemento
```

2.2.2. Errores comunes

El lenguaje C no comprueba las posiciones del array a las que intentamos acceder y, al igual que con el resto de variables, no inicializa la memoria al declarar un array. Esto puede producir errores de ejecución difíciles de encontrar:

- **Acceso a posiciones fuera del rango del array**: intentar acceder a un índice fuera del rango del array, como `numeros[5]` en un array de tamaño 5, genera un comportamiento indefinido, ya que la última posición válida en este caso es la posición 4. De igual manera, tampoco debemos acceder a elementos de un array con índices negativos, como `numeros[-1]`, por lo que debemos tener cuidado a la hora de recorrer nuestro array con estructuras de repetición. Si intentamos acceder a una posición fuera del rango de un array, C intentará acceder a la posición de memoria como si perteneciese al array, lo que puede ocasionar errores de ejecución, accesos a variables incorrectas o accesos a zonas de memoria protegida.
- **No inicializar el array**: declarar un array sin inicializarlo hace que dicho array contenga los datos que hubiese en la memoria que se ha reservado para el array antes de ser reservada. Esto puede dar lugar a valores aleatorios si se intenta acceder a sus elementos antes de asignarles un valor.

2.3. Tamaño

El tamaño del array se especifica en el momento de su declaración y no se puede cambiar durante la ejecución del programa. Además, la longitud de un array en C debe establecerse en tiempo de compilación, lo que significa que su número de elementos debe ser una

constante o valor que el compilador pueda calcular en el momento de compilar el programa. Sin embargo, a partir de la versión C99 del estándar, se introdujeron los arrays de longitud variable (o VLA), que permiten establecer (pero en ningún caso modificar) el número de elementos de un array en tiempo de ejecución. Esto permite el uso de variables que cambien su valor durante la ejecución de nuestro programa para definir el número de elementos de nuestro array.

En cualquier caso, en C, se puede determinar el tamaño de un array usando la función `sizeof`. Esta función retorna el tamaño en bytes del array en memoria, por lo que, si queremos determinar el número de elementos de dicho array, bastará con dividir el tamaño en bytes del array bien por el tamaño del tipo de datos de dicho array, bien por el tamaño de uno de sus elementos.

2.3.1. Ejemplos

El siguiente ejemplo muestra cómo podemos obtener el número de elementos de un array utilizando el operador `sizeof` de dos formas distintas: en la primera, haciendo uso del tipo de datos de los elementos del array; en la segunda, utilizando el tamaño del primer elemento del array.

```
// código para calcular el tamaño de un array
int numeros[5];
int tam1 = sizeof(numeros) / sizeof(int);  // Obtener el número de
elementos utilizando el tipo de datos
int tam2 = sizeof(numeros) / sizeof(numeros[0]);  // Obtener el
número de elementos utilizando un elemento
printf("El tamaño del array es: %d o %d\n", tam1, tam2);
```

2.4. Inicialización y recorrido

Un array se puede inicializar en el momento de su declaración, proporcionando los valores entre llaves (`{}`) y separados por comas (`,`). Estos valores deben ser del mismo tipo que el array.

2.4.1. Ejemplos

asdfasdf

```
    int numeros[5] = {1, 2, 3, 4, 5};  // Inicialización de un array
con 5 elementos
    char vocales[5] = {'a', 'e', 'i', 'o', 'u'};  // Array de
caracteres
// Recorrer el array con un bucle
    for (int i = 0; i < 5; i++) {
        printf("%d ", numeros[i]);
        printf("%c ", vocales[i]);
    }
```

Si el tamaño del array es mayor que el número de elementos proporcionados, los elementos restantes se inicializan a cero automáticamente, como se muestra en el siguiente ejemplo:

```
int numeros[5] = {1, 2}; // Los otros 3 elementos se inicializan a 0
```

Si inicializamos el array, es posible no indicar el número de elementos, en cuyo caso se reservará memoria para el número de elementos incluidos en la inicialización. Por ejemplo, el siguiente ejemplo declara e inicializa un array de 4 elementos:

```
int numeros[] = {2, 4, 6, 8}; // Se reserva memoria para 4 elementos
```

2.5. Array parcialmente llenos

En algunos casos, se puede necesitar un array cuyo tamaño es mayor que la cantidad de datos que se tiene al inicio. Además, en función de las posiciones en las que se encuentren los datos válidos podemos distinguir dos escenarios:

1. Todos los elementos válidos se almacenan en las primeras posiciones del array. En este caso, podemos utilizar una variable de tipo `int` que indique cuántos elementos válidos hay actualmente en el array.
2. Los elementos válidos se almacenan en ciertas posiciones del array, pero estas no son consecutivas ni siguen ningún orden. En este caso, es posible asignar un valor "basura" que no sea válido en el contexto de nuestro programa (por ejemplo, un número negativo para una edad). De esta forma, es posible saber si se ha inicializado o no el elemento de una determinada posición evaluando su valor.

2.5.1. Ejemplos

En este ejemplo, aunque el tamaño del array `numeros` es `MAX` (`10`), solo se llenan los primeros 5 elementos, llevando la cuenta en la variable `contador`:

```
#include <stdio.h>
#define MAX 10

int main() {
    int numeros[MAX];
    int contador = 0;
    // Llenar parcialmente el array
    for (int i = 0; i < 5; i++) {
        numeros[i] = i + 1;  // Asignar valores al array
        contador++;
    }
    printf("El array tiene %d elementos\n", contador);
    return 0;
}
```

El siguiente ejemplo muestra un array con valores válidos únicamente en algunas posiciones. Para saber qué valores son válidos, el array se inicializa al comienzo del programa a `-1`, un número negativo que no tiene sentido en el contexto del programa (edades de los usuarios). A continuación, el programa solicita al usuario que inserte diferentes edades, indicando la posición del array donde desea almacenarlas.

En ese momento, el array tendrá elementos con valores válidos y elementos con valores no válidos, por lo que, si queremos trabajar únicamente con los valores válidos es necesario comprobar previamente el valor de la posición. En este caso, el programa del ejemplo imprime únicamente las edades que ha insertado el usuario, para lo que debe comprobar antes el valor almacenado. En caso de ser un `-1` (valor basura al que hemos inicializado todas las posiciones del array), el programa no mostrará el valor de esa posición.

```c
#include <stdio.h>
#define MAX 10

int main() {
    int edades[MAX];
    int posicion, edad;

    for (int i = 0; i < MAX; i++) {
        edades[i] = -1; // inicializamos todos los elementos a -1
    }

    do {
        printf("Dame posicion y edad.");
        printf("Si quieres dejar de insertar datos, indica una
posición negativa.\n");
        scanf("%d %d", &posicion, &edad);
        if (posicion >= 0) {
            edades[posicion] = edad;
        }
    } while (posicion >= 0);

    for (int i = 0; i < MAX; i++) {
        if (edades[i] != -1) {  // Solo se muestran aquellos valores
diferentes a -1
            printf("Edad de la posicion %d: %d\n", i, edades[i]);
        }
    }
    return 0;
}
```

3. Arrays bidimensionales

Los arrays bidimensionales en C son estructuras de datos que permiten almacenar información en una tabla de filas y columnas, es decir, en una matriz. Se declaran especificando dos índices: uno para las filas y otro para las columnas, y cada elemento de la matriz es accesible mediante su posición en ambos. También es posible ver estos arrays bidimensionales como "arrays de arrays", en los que el array bidimensional es un array compuesto de múltiples arrays, cada uno compuesto de diferentes elementos de un determinado tipo. Este tipo de array es útil para representar datos que tienen una estructura tabular, como tablas de números, matrices matemáticas o incluso imágenes.

Los arrays bidimensionales facilitan el manejo de datos relacionados en dos dimensiones y son esenciales para algoritmos complejos que requieren múltiples niveles de organización.

3.1. Sintaxis

La sintaxis básica de un arreglo bidimensional es:

```
tipo id_matriz[num_filas][num_columnas];
```

Donde:

- `tipo`: es el tipo de datos de los elementos del array bidimensional o matriz (por ejemplo, `int`, `float`, o `char`).

- `id_matriz`: es el nombre de la matriz. Debemos utilizar un identificador válido según las reglas detalladas en el Tema 2.
- `num_filas`: es el número de filas que el array bidimensional puede contener. También puede verse como el número de arrays que este "array de arrays" contiene.
- `num_columnas`: es el número de columnas que contiene el array bidimensional. Si consideramos a una matriz como un "array de arrays", sería el número de elementos que contiene cada uno de los arrays.

Así, un array bidimensional definido como `int matriz[N][M]` tendrá un total de `N*M` elementos. Por ejemplo, una matriz de 3 filas y 4 columnas tendrá un total de 12 elementos.

3.2. Almacenamiento de datos de un array bidimensional en memoria

Un array bidimensional se almacena en la memoria del ordenador como una secuencia lineal y continua de datos. Esto significa que todos los elementos del array se guardan uno tras otro, sin espacios entre ellos, siguiendo un orden determinado.

En C, el almacenamiento se realiza en lo que se conoce como "Row-Major Order" o Filas Principales. Esto quiere decir que todos los elementos de un array bidimensional se almacenan en memoria de manera consecutiva y fila a fila, almacenándose todos los elementos de una determinada fila consecutivamente antes de pasar a la siguiente fila.

Así pues, la siguiente matriz:

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

Se almacenaría en memoria de la siguiente manera:

- Primero se almacenan todos los elementos de la primera fila: `{1, 2, 3}`.
- Luego se almacenan los elementos de la segunda fila `{4, 5, 6}`.
- Finalmente, los elementos de la tercera fila `{7, 8, 9}`.

Quedando en memoria como: `{1, 2, 3, 4, 5, 6, 7, 8, 9}`.

3.3. Cálculo de la dirección de un elemento

Al almacenarse en memoria todos los elementos de manera consecutiva y por filas, es posible acceder a la dirección de memoria de cada elemento individual de un array bidimensional `matriz[i][j]` calculando su dirección de memoria como:

```
dirección_base + (i * número_columnas + j) * tamaño_tipo
```

Donde:

- `dirección_base`: es la dirección de memoria del primer elemento del array (*matriz[0][0]*).
- `i`: es el índice de la fila.
- `número_columnas`: es la cantidad de columnas del array.
- `j`: es el índice de la columna.
- `tamaño_tipo`: es el tamaño en bytes del tipo de datos del array (por ejemplo, 4 bytes para *int* en la mayoría de las arquitecturas).

En el ejemplo anterior:

- `matriz[0][0]` se almacena en la dirección base.
- `matriz[1][1]` se encuentra en la `dirección_base` desplazada por `(1 * 3 + 1) * tamaño_del_tipo`, que apunta al valor *5*.

3.4. Ventaja del almacenamiento lineal

El almacenamiento contiguo en memoria permite un acceso rápido a los elementos del array, ya que es posible calcular la dirección de cualquier elemento conociendo solo la dirección base y los índices, sin necesidad de almacenar punteros adicionales. Esto optimiza el uso de memoria y mejora la eficiencia en la manipulación de datos.

Este modelo de almacenamiento es particularmente útil para operaciones matemáticas y computacionales que involucren matrices, ya que facilita recorrer filas de manera secuencial.

4. Array multidimensionales

Los arrays multidimensionales son estructuras que permiten almacenar datos en múltiples dimensiones, lo cual resulta útil para representar tablas, matrices, o cualquier estructura que requiera más de una dimensión.

Un array multidimensional es una extensión del array unidimensional, donde cada elemento del array puede ser otro array, permitiendo el almacenamiento de datos en forma de tablas o matrices. El más comúnmente utilizado es el array bidimensional, que se asemeja a una matriz o tabla de filas y columnas.

Por ejemplo, un array bidimensional `int matriz[3][4]` representa una matriz de 3 filas y 4 columnas donde se almacenan 12 elementos enteros. Este array bidimensional también puede verse como un array de 3 elementos, siendo cada uno de ellos un array de 4 elementos de tipo entero. Así pues, de manera general, un array de `n` dimensiones se puede visualizar como un conjunto de array anidados.

4.1. Sintaxis

Para declarar un array multidimensional, se usa la siguiente sintaxis en C:

$$\texttt{tipo id\_array[num\_El}_1\texttt{][num\_El}_2\texttt{]...[num\_El}_N\texttt{];}$$

- `tipo`: el tipo de datos que almacenará el array (por ejemplo*, int, float, char*).
- `id_array`: el nombre del array.
- $\texttt{num\_El}_1, \texttt{num\_El}_2, \ldots, \texttt{num\_El}_N$: los tamaños de cada dimensión del array.

4.1.1. Ejemplos

A continuación, se muestra un programa que declara un array bidimensional de enteros con 3 filas y 4 columnas. Como puede verse en este ejemplo, la declaración incluye el número de elementos en cada una de las dimensiones. Al igual que sucedía con los arrays unidimensionales, podemos utilizar los caracteres llaves (`{}`) para inicializar los valores de los elementos. Al tratarse de un array bidimensional, estos elementos se inicializan como arrays de arrays. Finalmente, el ejemplo también accede a cada uno de los elementos del

array para mostrarlos por pantalla. En este caso, al tratarse de un array bidimensional, se utilizan dos bucles `for` anidados para recorrerlo: uno para las filas y otro para las columnas. Por norma general, utilizaremos un bucle por cada dimensión del array.

```c
#include <stdio.h>
int main() {
    // Declaración e inicialización de un array bidimensional
    int matriz[3][4] = {
            {1, 2, 3, 4},
            {5, 6, 7, 8},
            {9, 10, 11, 12}
    };
    // Acceder a los elementos del array y mostrarlos
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 4; j++) {
            printf("matriz[%d][%d] = %d\n", i, j, matriz[i][j]);
        }
    }
    return 0;
}
```

También es posible trabajar con array de tres o más dimensiones. El siguiente ejemplo muestra la declaración de un array tridimensional, que representa un cubo de dimensión de 2 x 3 x 4. En este caso, la declaración `int cubo[2][3][4]` se utiliza para crear un array tridimensional, indicando el número de elementos de cada dimensión. Cada valor se puede visualizar como una "capa" del cubo (2 capas), cada capa con 3 filas y 4 columnas, dando así un cubo de 24 elementos. Al igual que pasaba en el ejemplo anterior, podemos recorrer el cubo utilizando tres bucles `for` anidados, uno para cada dimensión.

```c
#include <stdio.h>
int main() {
  int cubo[2][3][4] = { // Declaración e inicialización
    {
      {1, 2, 3, 4},
      {5, 6, 7, 8},
      {9, 10, 11, 12}
    },
    {
      {13, 14, 15, 16},
      {17, 18, 19, 20},
      {21, 22, 23, 24}
    }
  };
    // Acceder a los elementos del array tridimensional y mostrarlos
  for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++) {
      for (int k = 0; k < 4; k++) {
        printf("cubo[%d][%d][%d] = %d\n", i, j, k, cubo[i][j][k]);
      }
    }
  }
  return 0;
}
```

5. Cadenas de caracteres

En C, una cadena de caracteres es una secuencia de caracteres almacenada en un array y que se encuentra terminada con un carácter nulo (`'\0'`). Este tipo de dato es esencial para trabajar con textos, ya que permite almacenar y manipular secuencias de caracteres.

Este conjunto de caracteres puede ser tratado como un único dato. Se almacena en un array de tipo `char` y siempre termina con un carácter nulo (`'\0'`) que indica el final de la cadena. Esto es necesario para que las funciones que operan sobre cadenas sepan dónde finaliza la cadena (no es lo mismo una cadena de caracteres que un array de caracteres).

Por ejemplo: `"Hola"` es una cadena que internamente está representada como un array con los cuatro caracteres más el carácter nulo: `{'H', 'o', 'l', 'a', '\0'}`.

5.1. Sintaxis

Para declarar una cadena de caracteres en C, se utiliza un array de tipo `char`:

```
char id_cadena [num_elementos];
```

Donde:

- char: todas las cadenas de caracteres deben ser de este tipo.
- id_cadena: nombre que daremos a la cadena de caracteres.
- num_elementos: número de caracteres que contendrá la cadena. No debemos olvidar contabilizar el carácter nulo (`'\0'`).

5.1.1. Ejemplo

El siguiente ejemplo muestra la declaración de dos cadenas de caracteres, el primero de ellos indicando el número de elementos y sin inicializar, mientras que en el segundo caso no se indica el número de elementos. Al igual que sucede con los arrays de otros tipos, en este caso el compilador reservará espacio en memoria para la cadena indicada en la inicialización.

```
// declaración de un array de 20 caracteres, capaz de almacenar una
cadena de hasta 19 caracteres más el carácter nulo ('\0'):
char cadena[20];
// declaración donde el compilador agrega automáticamente el
carácter nulo al final de la cadena.
char saludo[] = "Hola";
```

5.2. Inicialización

La inicialización de una cadena de caracteres puede hacerse directamente con un literal (más común) o carácter a carácter:

5.2.1. Directa con un literal de cadena

En este caso se asocia directamente el valor de una cadena de caracteres a la cadena declarada. Esta asociación solamente puede hacerse en el momento de su declaración:

```
// Declaración de la cadena a la que se le asigna el valor "Hola
Mundo". El tamaño del array se ajusta automáticamente a 11 elementos
char saludo[] = "Hola Mundo";
```

5.2.2. Inicialización carácter por carácter

Otra opción es asignar los valores de cada elemento de manera individual, incluyendo los valores entre llaves (`{}`), al igual que se hace con el resto de arrays:

```
// En esta inicialización se especifican explícitamente los
caracteres de la cadena, incluyendo el carácter nulo '\0'
char saludo[6] = {'H', 'o', 'l', 'a', '\0'};
```

5.3. Lectura y escritura de cadenas de caracteres

Para trabajar con cadenas de caracteres en C, podemos utilizar varias funciones, como `scanf` y `printf` (que son parte de la librería `<stdio.h>` presentada en el Tema 2) o funciones de la librería `<string.h>` (que veremos en una sección siguiente). A continuación, se presentan algunos ejemplos:

5.3.1. Lectura con scanf

En el Tema 2 se introdujo la entrada y salida estándar en C, presentando la función `scanf`. Esta función permite leer cadenas de caracteres con `%s`. Sin embargo, se presentan dos inconvenientes, el primero es que `%s` no especifica el número máximo de caracteres a leer, lo que puede provocar problemas si intentamos almacenar más caracteres de los reservados en memoria al declarar la cadena de caracteres. El segundo problema es que, utilizando `%s`, `scanf` únicamente lee hasta que encuentra un espacio o salto de línea, impidiendo leer cadenas con espacios.

Para establecer el máximo de caracteres a leer es posible indicar este número en el código de formato, entre `%` y `s`. Así, el siguiente ejemplo especifica el número máximo de elementos para evitar el desbordamiento del buffer, reservando una posición para el carácter nulo:

```
#include <stdio.h>

int main() {
    char nombre[50];
    printf("Introduce tu nombre: ");
    scanf("%49s", nombre); // Lee una cadena sin espacios
    printf("Hola, %s\n", nombre);
    return 0;
}
```

Así mismo, también es posible sustituir `%s` por `%[^\n]`, una expresión regular que indica que se lea cualquier carácter diferente al carácter `'\n'`. En el siguiente ejemplo la función `scanf` únicamente se detendrá cuando encuentre un salto de línea (`'\n'`), no cualquier espacio:

```
#include <stdio.h>

int main() {
    char nombre[50];
    printf("Introduce tu nombre: ");
    scanf("%[^\n]", nombre); // Lee una cadena con espacios
    printf("Hola, %s\n", nombre);
    return 0;
}
```

5.3.2. Lectura con fgets

Otra opción es utilizar la función `fgets`, la cual tiene la siguiente sintaxis:

```
fgets(cad, num_elem, stdin);
```

Y permite almacenar en la cadena `cad` los `num_elem-1` caracteres leídos de la entrada estándar (`stdin`). Esta función lee hasta alcanzar los `num_elem-1` caracteres o hasta un salto de línea (`'\n'`). Al obligarnos a indicar el número máximo de caracteres a leer, esta función se considera segura para leer cadenas de caracteres.

A continuación, se muestra un ejemplo del uso de `fgets`, indicando el número de elementos de la cadena frase como número máximo de caracteres a leer:

```c
#include <stdio.h>

int main() {
    char frase[100];
    printf("Introduce una frase: ");
    fgets(frase, sizeof(frase), stdin); // Lee una cadena con
espacios
    printf("La frase introducida es: %s\n", frase);
    return 0;
}
```

5.3.3. Escritura con printf

De igual manera, es posible utilizar el código de control `%s` para mostrar una cadena de caracteres con la función `printf`:

```c
#include <stdio.h>

int main() {
    char saludo[] = "Hola, mundo!";
    printf("%s\n", saludo); // Escribe la cadena en la consola
    return 0;
}
```

5.3.4. Otras funciones de entrada y salida

Aunque las funciones descritas anteriormente son las más utilizadas para lectura y escritura de cadenas de caracteres, existen otras funciones que permiten trabajar con estas cadenas. Por ejemplo, `puts()` y `fputs()` escriben por consola la cadena de caracteres indicada como parámetro, incluyendo un salto de línea.

Por otro lado, la función `gets()` también permite leer una cadena de caracteres hasta encontrar un salto de línea, pero no permite especificar el número máximo de caracteres a leer.

Otras funciones como `getchar()` permiten leer un carácter de teclado cuando el usuario pulsa intro (salto de línea).

6. Librería string.h

La librería `string.h` proporciona una serie de funciones para manipular y operar con cadenas de caracteres en C. A continuación, se explican algunas de las funciones más utilizadas de esta librería: `strcpy`, `strlen`, `strcmp` y `strcat`.

6.1. strcpy

En C, no es posible copiar una cadena de caracteres en otra utilizando el operador de asignación (=). Por ello, es necesario hacer una copia carácter a carácter, o utilizar una función para ello. La función `strcpy` se utiliza para **copiar una cadena de caracteres** en otra. Su prototipo es:

```
char *strcpy(char *destino, const char *origen);
```

Donde sus parámetros son:

- `destino`: cadena donde se copiará el contenido.
- `origen`: cadena que se copiará en destino.

6.1.1. Ejemplo

El siguiente ejemplo hace uso de la función `strcpy(destino, origen)` para copiar la cadena origen en destino. Es importante asegurarse de que destino tenga suficiente espacio para almacenar la cadena origen y el carácter nulo (`'\0'`), ya que la función no lo hace de manera automática.

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

int main() {
    char origen[] = "Hola, mundo!";
    char destino[20];
    strcpy(destino, origen);
    printf("Destino: %s\n", destino); // Salida: "Destino: Hola,
mundo!"
    return 0;
}
```

6.2. strlen

La función `strlen` se utiliza para calcular la longitud de una cadena, excluyendo el carácter nulo (`'\0'`). Dicho de otra forma, esta función retorna la posición del carácter nulo en el array de caracteres. Su prototipo es:

```
size_t strlen(const char *cadena);
```

Donde su parámetro es:

- `cadena`: la cadena de la cual se desea conocer la longitud.

6.2.1. Ejemplo:

El siguiente ejemplo hace uso de la función `strlen(texto)` para obtener y almacenar la longitud de la cadena de texto, que en este caso es 17 (no cuenta el carácter nulo). Ten en

cuenta que esta longitud no coincide con el número de elementos del array, que en este caso es 18, al ser el último carácter el carácter nulo (`'\0'`).

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

int main() {
    char texto[] = "Programación en C";
    int longitud = strlen(texto);
    printf("La longitud de la cadena es: %d\n", longitud); //
Salida: "La longitud de la cadena es: 17"
    return 0;
}
```

6.3. strcmp

Al igual que pasaba con la asignación, el operador de comparación (==) no permite comparar cadenas de caracteres. En este caso, podemos comprobar si dos cadenas son iguales comparando cada uno de sus caracteres de manera individual, tal y como lo haríamos con un array. Otra opción es utilizar la función `strcmp`. Su prototipo es:

```c
int strcmp(const char *cadena1, const char *cadena2);
```

Donde los parámetros:

- `cadena1` y `cadena2`: las cadenas a comparar.

Esta función retorna diferentes valores:

- Un valor negativo si `cadena1` es menor alfabéticamente que `cadena2`.
- 0 si ambas cadenas son iguales.
- Un valor positivo si `cadena1` es mayor alfabéticamente que `cadena2`.

6.3.1. Ejemplo:

El siguiente ejemplo hace uso de la función `strcmp` para comparar cadenas. En este caso, `cadena1` y `cadena2` son iguales, por lo que la comparación devuelve 0. La comparación entre `cadena1` y `cadena3` devuelve un valor distinto de 0 porque son diferentes:

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

int main() {
    char cadena1[] = "Hola";
    char cadena2[] = "Hola";
    char cadena3[] = "Mundo";
    if (strcmp(cadena1, cadena2) == 0) {
        printf("Las cadenas 1 y 2 son iguales.\n");
    } else {
        printf("Las cadenas 1 y 2 son diferentes.\n");
    }
    if (strcmp(cadena1, cadena3) != 0) {
        printf("Las cadenas 1 y 3 son diferentes.\n");
    }
    return 0;
}
```

6.4. strcat

La función `strcat` se utiliza para concatenar (unir) dos cadenas de caracteres. Su prototipo es:

```
char *strcat(char *destino, const char *origen);
```

Donde sus parámetros son:

- `destino`: cadena a la cual se le añadirá el contenido de `origen`. A diferencia de la función `strcpy`, en este caso el contenido actual de la cadena `destino` no se elimina.
- `origen`: cadena que se añadirá a *destino*.

6.4.1. Ejemplo:

El siguiente ejemplo hace uso de la función `strcat(saludo, complemento)` para añadir el contenido de la cadena `complemento` al final de la cadena `saludo`. Es importante asegurarse de que la cadena `destino` (`saludo` en este caso) tenga suficiente espacio reservado en memoria para almacenar la cadena resultante.

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

int main() {
    char saludo[20] = "Hola";
    char complemento[] = ", mundo!";
    strcat(saludo, complemento);
    printf("Cadena concatenada: %s\n", saludo); // Salida: "Cadena
concatenada: Hola, mundo!"
    return 0;
}
```

Tema 6: Punteros

Cuando declaramos una variable, el sistema reserva una región de memoria para almacenar un dato del tipo indicado en su declaración. Después, cada vez que utilizamos el identificador de dicha variable, estamos accediendo a la dirección de memoria que tiene asignada. Dicho de otra forma, cuando accedemos al valor de una variable, estamos accediendo a su dirección de memoria.

En C, podemos trabajar con estas direcciones de memoria. Entre otras cosas, es posible almacenarlas en variables denominadas punteros, los cuales se presentan a lo largo de este tema.

1. Definición y usos

A diferencia de las variables tradicionales, que contienen un valor de un tipo determinado, un puntero es una variable que nos permite almacenar una dirección de memoria. Esta dirección de memoria puede ser la dirección de otra variable o una dirección de memoria reservada dinámicamente. La reserva de memoria dinámica se detallará en el Tema 8.

Cuando un puntero contiene una dirección de memoria válida se dice que "apunta" a dicha dirección. Así, si un puntero tiene almacenada la dirección de memoria de una variable, podemos decir que dicho puntero "apunta" a esa variable.

Los punteros resultan de gran utilidad en C, ya que permiten manipular y acceder a valores ubicados en diferentes lugares de memoria, lo que es indispensable para el paso de parámetros por referencia (que se detalla en el Tema 7) y la gestión de la memoria dinámica. Además, permite manejar de manera eficiente arrays y matrices, así como optimizar ciertos algoritmos.

2. Declaración

La declaración de un puntero es similar a la de otra variable, pero incluye un asterisco que indica que se trata de un puntero. Así, la declaración de un puntero es:

```
tipo * id_puntero;
```

Donde:

- `tipo` indica el tipo del dato al que apuntará el puntero. También puede verse como el tipo de dato almacenado en la dirección de memoria que contendrá el puntero.
- `*` indica que la variable es un puntero.
- `id_puntero` es el identificador de la variable. Estos identificadores siguen las normas establecidas para el resto de identificadores en C.

El siguiente ejemplo muestra la declaración de punteros que apuntarán a valores de diferentes tipos: enteros, punto flotante y carácter.

```
int * pInt;     // Puntero a un valor entero.
float * pFloat; // Puntero a un valor en punto flotante.
char * pChar;   // Puntero a un valor char.
```

©2024 Diego Hortelano Haro, Gerardo Reyes Salgado, Manuel Rubio Sánchez. Algunos derechos reservados. Este documento se distribuye bajo la licencia "Atribución/Reconocimiento-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en https://creativecommons.org/licenses/by-sa/4.0/deed.es.

3. Puntero a NULL

Al igual que sucede con el resto de variables, C no inicializa el contenido de los punteros. Esto puede provocar errores al intentar acceder a regiones de memoria que no hemos reservado. Por ello, es importante inicializar nuestros punteros. Si deseamos indicar que un puntero está "vacío", podemos apuntarlo a la constante `NULL`, la cual representa un puntero nulo o que no apunta a ninguna dirección válida. Para ello, es suficiente con asignarle dicho valor:

```
int * p = NULL;
```

Sin embargo, si intentamos acceder a la dirección de este puntero obtendremos un error de ejecución. Por ello, es importante comprobar si un puntero contiene una dirección de memoria válida antes de acceder a él si no estamos seguros de que dicha dirección sea válida. Si hemos inicializado el puntero a `NULL`, es suficiente con comprobar si el valor que contiene sigue siendo `NULL`:

```
if(p != NULL)
{
    printf("El puntero contiene una dirección de memoria válida\n");
}else{
    printf("El puntero contiene una dirección de memoria NO
válida\n");
}
```

De esta forma podremos evitar acceder a direcciones de memoria erróneas o a punteros no inicializados.

4. Puntero a puntero

Como se ha expuesto, los punteros son otro tipo de variable, por lo que tienen su propia dirección de memoria. Por ello, si un puntero contiene un valor de tipo dirección de memoria, sería posible que este valor fuese la dirección de otro puntero. Dicho de otra manera, un puntero "apunta" a una variable cualquiera, pudiendo ser dicha variable otro puntero. Este tipo de punteros se denomina puntero a puntero, y es necesario indicarlo en la declaración, teniendo en cuenta que el valor apuntado en este caso será una dirección de memoria a un tipo de dato. En su declaración se incluye un doble asterisco:

```
tipo ** id_puntero;
```

Donde:

- `tipo` indica el tipo del valor apuntado. En este caso, al tratarse de un puntero a puntero, representa el tipo del dato apuntado por la dirección de memoria a la que apunta el contenido de dicho puntero (dos niveles).
- `**` indica que la variable es un puntero a puntero.
- `id_puntero` es el identificador de la variable. Estos identificadores siguen las normas establecidas para el resto de identificadores en C.

En C no hay un límite definido en lo referente a la cantidad de capas de indireccionamiento (o niveles de punteros) que podemos tener (este límite lo fijará la memoria del sistema). Por lo tanto, podemos tener tantos niveles de punteros como queramos, aunque en la práctica

es extraño encontrar más de tres niveles. Estos niveles siguen las mismas reglas vistas hasta ahora. El siguiente ejemplo muestra la declaración de punteros de hasta tercer nivel:

```
    int var = 10;
    int *p1 = &var;      // Puntero de primer nivel
    int **p2 = &p1;      // Puntero de segundo nivel
    int ***p3 = &p2;     // Puntero de tercer nivel
```

5. Operadores

Aunque hay diferentes operadores que podemos utilizar con punteros (los cuales se presentan más adelante), destacan especialmente los operadores `&` y `*`.

El operador `&` (denominado operador de dirección u operador de referencia) permite obtener la dirección de memoria de una variable. Es especialmente útil para almacenar en punteros las direcciones de memoria de las variables utilizadas en nuestros programas.

Por su parte, el operador `*` (denominado operador de indirección u operador de desreferenciación) permite acceder al dato almacenado en una dirección de memoria. Este operador permite obtener el valor contenido en la dirección de memoria apuntada por un puntero para utilizarlo o modificarlo.

Ambos operadores se utilizan a la izquierda de la variable utilizada como operando, como se muestra en el siguiente ejemplo. El programa del ejemplo comienza declarando tres variables, una variable entera (`var`), un puntero a un entero (`p1`), un puntero a puntero (`p2`) y un puntero de tercer nivel (`p3`). En esta declaración se inicializan las variables, utilizando el operador `&` para obtener la dirección de memoria de la variable o puntero apuntado. La segunda parte del programa hace uso del operador `*` para acceder a los datos apuntados:

```
#include <stdio.h>

int main() {
    int var = 10;
    int *p1 = &var;      // Puntero de primer nivel
    int **p2 = &p1;      // Puntero de segundo nivel
    int ***p3 = &p2;     // Puntero de tercer nivel

    // Accediendo al valor de var a través de diferentes punteros:
    // Acceso directo a var:
    printf("Valor de x = %d\n", var);
    // Usando el puntero de primer nivel:
    printf("Valor de x usando *p1 = %d\n", *p1);
    // Usando el puntero de segundo nivel
    printf("Valor de x usando **p2 = %d\n", **p2);
    // Usando el puntero de tercer nivel:
    printf("Valor de x usando ***p3 = %d\n", ***p3);

    return 0;
}
```

Es necesario destacar que, como se muestra en el ejemplo anterior, es posible encadenar el operador `*` múltiples veces para acceder a varios niveles de indireccionamiento. Sin embargo, no es posible utilizar de esta forma el operador `&`, ya que este operador solo se aplica a variables, las cuales tienen una única dirección de memoria, y no es posible

obtener la dirección de una dirección. Por lo tanto, el operador `&` solo puede aplicarse una vez sobre una variable.

Otro punto a destacar aquí es que, como ya hemos comentado, los punteros son variables, por lo que pueden cambiar su valor a lo largo de la ejecución del programa, cambiando la variable o región de memoria a la que apuntan. Para ello basta con cambiar el contenido del puntero, como muestra el siguiente ejemplo:

```
int var1, var2;
int *p;
p = &var1;  // Apuntamos p a var1, almacenando su dirección.
*p = 10;    // Modificamos var1 a través del puntero p.
p = &var2;  // Apuntamos p a var2, almacenando la dirección de var2.
*p = 20;    // Modificamos var2 a través del puntero p.
```

Es importante prestar atención al uso de los operadores `*` y `&`. Cuando se desea cambiar el contenido del puntero, es decir, la dirección de memoria que almacena (o apunta), no se debe utiliza el operador `*` en la parte izquierda del operador de asignación. En este caso, en la parte derecha del operador de asignación debe colocarse una dirección de memoria, por lo que usualmente se utiliza el operador `&`. Por otro lado, si queremos modificar el valor almacenado en la dirección de memoria apuntada por el puntero, debemos utilizar el operador `*` en la parte izquierda del operador de asignación. En este caso, el valor de la parte derecha del operador de asignación debe ser del mismo tipo de dato que el contenido apuntado.

6. Errores comunes

Al trabajar con punteros es sencillo cometer errores difíciles de detectar. Para evitar estos errores debemos tener claro los siguientes puntos:

1. No es posible cambiar la dirección de una variable. Cuando declaramos una variable, el sistema les asigna una dirección de memoria que mantendrá durante la ejecución del programa. Por ello, nunca debemos incluir el operador `&` en la parte izquierda de un operador de asignación. Por ejemplo, `&i=p;` es una sentencia incorrecta, ya que la dirección de la variable `i` no puede ser modificada.
2. No debemos asignar una dirección absoluta o un entero a un puntero directamente. Las direcciones de memoria utilizadas por un puntero deben ser válidas, por lo que asignar manualmente una dirección de memoria de manera arbitraria es una práctica peligrosa. Si la dirección no es válida o accesible, la ejecución de nuestro programa provocará errores de acceso. En su lugar, es preferible utilizar direcciones asignadas por el sistema a variables o a regiones de memoria dinámica.
3. No se pueden hacer asignaciones directas entre punteros que apuntan a diferentes tipos de datos sin una conversión explicita (casting). Es importante recordar que la dirección almacenada en un puntero se interpreta según el tipo de dato al que apunta. Por ejemplo, si se copia la dirección almacenada en un puntero de tipo `int *` a otro de tipo `float *`, al acceder a través del puntero `float *` se estaría interpretando un valor de tipo entero como si fuera un valor de tipo punto flotante, produciendo resultados incorrectos.

7. Punteros void

Los punteros de tipo `void *` son punteros genéricos, por lo que podemos asignarles cualquier tipo de puntero o hacerlos apuntar a variables de cualquier tipo. Esto es especialmente útil para trabajar con punteros sin especificar su tipo exacto. En su declaración se incluye el tipo genérico (`void`) en lugar del tipo de dato apuntado. El siguiente ejemplo muestra la declaración de un puntero genérico y la asignación de direcciones que contienen valores de diferentes tipos, tanto a través de la dirección de las variables como a través de otros punteros:

```
int vInt = 3, *pInt;
double vDouble = 8.2, *pDouble;
void * pVoid;

pInt = &vInt;
pDouble = &vDouble;

// Utilizando directamente la dirección de una variable:
pVoid = &vInt;    // El puntero genérico apunta a un entero.
pVoid = &vDouble; // El puntero genérico apunta a un punto flotante.

// Utilizando el contenido de otro puntero:
pVoid = pInt;     // El puntero genérico apunta a un entero.
pVoid = pDouble;  // El puntero genérico apunta a un punto flotante.
```

Este tipo de punteros tiene una gran flexibilidad, por lo que son de gran ayuda cuando tenemos una función o estructura que deba trabajar con diferentes tipos de datos que son desconocidos de antemano. Por ello, su uso es común en funciones genéricas, como en la función estándar `malloc`, que permite reservar memoria dinámica de cualquier tipo, retornando un puntero genérico. Esta función se detallará en el Tema 8. Sin embargo, este tipo de punteros no pueden ser desreferenciados directamente, ya que no tienen un tipo de dato definido. Para ello, es necesario convertirlo a un tipo de dato de manera explícita, como se muestra en el siguiente ejemplo:

```
int vInt = 3, *pInt;
void * pVoid;

// Apuntamos pInt y pVoid a la misma variable:
pInt = &vInt;
pVoid = &vInt;

// El puntero pInt puede desreferenciarse directamente:
printf("%d\n", *pInt);

// El puntero genérico pVoid debe desreferenciarse con una
conversión:
printf("%d\n", *(int *)pVoid);
```

En caso de omitir la conversión explicita o casting, obtendremos un error de compilación y no podremos ejecutar nuestro programa. Además, este tipo de punteros puede ocasionar errores difíciles de detectar. Por ejemplo, el siguiente bloque de código asigna la dirección de una variable de tipo entero al puntero genérico `pVoid`. Después, copia esa dirección de memoria en el puntero de tipo punto flotante de doble precisión `pDouble`, intentando mostrar el valor del tipo apuntado. En este caso, la conversión no se realiza correctamente,

mostrando un valor erróneo, ya que no es posible utilizar un puntero de un determinado tipo para acceder a una dirección de memoria que contiene un valor de un tipo diferente.

```
int vInt = 3;
void * pVoid;
double * pDouble;

// Apuntamos pVoid a una variable de tipo entero:
pVoid = &vInt;

// Apuntamos pDouble a la dirección de memoria almacenada en pVoid
pDouble = pVoid;

// Intentamos acceder al dato apuntado por pDouble:
printf("%lf\n", *pDouble);
```

8. Escritura de direcciones de memoria

Es posible utilizar la función `printf()` para mostrar por pantalla una dirección de memoria (lo que incluye el contenido de un puntero). Para ello, es necesario incluir el código de control "`%p`" en la cadena de control de la función, así como la dirección de memoria a mostrar tras la cadena de control. La dirección de memoria mostrada puede estar almacenada en un puntero o extraerse directamente utilizando el operador de dirección (`&`). El siguiente ejemplo imprime la dirección de la variable `var` utilizando el operador `&` y accediendo al valor contenido en el puntero `pVar`:

```
int var = 2;
int * pVar = &var;
printf("Dirección de var: %p. Valor de var: %d\n", &var, var);
printf("Dirección de pVar: %p. Valor de pVar: %p\n", &pVar, pVar);
```

Este ejemplo mostrará una salida similar a la siguiente:

```
Dirección de var: 00000045e49ff8ec. Valor de var: 2
Dirección de pVar: 00000045e49ff8e0. Valor de pVar:
00000045e49ff8ec
```

Es necesario resaltar aquí que la dirección de `var` y el contenido de `pVar` deben ser iguales (ya que el puntero `pVar` apunta a la variable `var`), mientras que el puntero `pVar` está almacenado en su propia dirección de memoria, diferente a la del resto de variables.

9. Aritmética de punteros

Es posible utilizar algunos operadores aritméticos con punteros. Estos operadores son la suma (+), la resta (−) y los operadores de incremento (++) y decremento (−−). Estas operaciones permiten modificar la dirección de memoria almacenada en un array. Sin embargo, el valor almacenado no se modifica exactamente en el indicado en la operación, si no que se tiene en cuenta el tamaño del tipo de dato apuntado. Así, si incrementamos en `N` una dirección de memoria, su valor no se incrementará en `N` bytes, sino que lo hará en `N` elementos del tipo apuntado. Dicho de otra forma, se modificará en `N*sizeof(tipo)` bytes. Esto es especialmente útil a la hora de trabajar con direcciones de memoria consecutivas, como los arrays.

9.1. Incremento y decremento de punteros

Como se ha comentado antes, es posible modificar el valor almacenado en un puntero (la dirección de memoria apuntada) incrementándola y decrementándola. En este caso, su valor variará en función del tamaño del tipo de datos apuntado. Esto es especialmente útil para desplazarnos por elementos del mismo tipo que se almacenen consecutivamente en memoria, como sucede con los arrays. De esta forma, es posible apuntar un puntero al primer elemento de un array y trabajar con los diferentes elementos modificando la dirección apuntada, como muestra el siguiente ejemplo:

```
int array[5] = {1, 2, 3, 4, 5};

// Creamos un puntero que apunte al primer elemento del array:
int * pArray = array;

// Utilizamos el puntero pArray para acceder al valor de los
diferentes elementos.
// Para ello, es necesario ir incrementando su valor:
for(int i = 0; i < 5; i++)
{
    printf("%d\t", *pArray);
    pArray++;
}
```

9.2. Diferencia de punteros

Un caso particular de las operaciones con punteros es la diferencia de punteros, ya que permite obtener la distancia que separa dos direcciones de memoria. Al igual que sucede con el resto de operaciones, la diferencia de punteros tiene en cuenta el tamaño del tipo apuntado, por lo que únicamente es posible restar punteros del mismo tipo. Por ello, el resultado de esta operación es el número de elementos que separan las dos direcciones, no el número de bytes. Por ejemplo, es posible utilizar la diferencia de punteros para obtener el número de elementos de un array:

```
int array[5] = {1, 2, 3, 4, 5};

// Creamos un puntero que apunte al primer elemento del array:
int * pInicial = array;
// Creamos un puntero que apunte al último elemento del array:
int * pFinal = &array[4];

// Mostramos el número de elementos:
printf("Número de elementos: %d\n", pFinal - pInicial + 1);
```

También es posible utilizar esta información para recorrer un array. El siguiente ejemplo utiliza dos punteros para mostrar los valores almacenados en un array:

```
int array[5] = {1, 2, 3, 4, 5};
// Creamos un puntero que apunte al primer elemento del array:
int * pInicial = array;
// Creamos un puntero que apunte al último elemento del array:
int * pFinal = &array[4];
// Mostramos cada uno de los elementos:
while(pInicial <= pFinal){
    printf("%d\t", *pInicial);
    pInicial++;
}
```

10.Arrays y punteros

En C, un array se comporta de manera similar a un puntero que contiene la dirección de la zona de memoria reservada para un determinado número de elementos del mismo tipo. Así, el identificador del array es, en esencia, una referencia a la dirección de memoria del primer elemento del array. Sin embargo, este puntero es un puntero constante, cuya dirección de memoria apuntada es inmutable (siempre representa la dirección del primer elemento).

10.1. Acceso

En el Tema 5 hemos visto cómo acceder a los diferentes elementos de un array utilizando indexación de arrays, indicando la posición del elemento entre corchetes (`[]`) junto a su identificador. Ahora, también podemos utilizar punteros para acceder a los elementos de un array. Esto quiere decir que es posible acceder al primer elemento de un array utilizando el índice 0 (`array[0]`) y accediendo al valor apuntado por la dirección de memoria del array con el operador de desreferencia (`*array`). Generalizando al resto de casos, tenemos dos opciones para acceder a cada elemento de un array: utilizando indexación (`array[posición]`) y utilizando el operador de desreferencia y la aritmética de punteros (`*(array+posición)`). De nuevo, es necesario recalcar aquí que esto es posible únicamente porque los arrays se almacenan de manera consecutiva en C.

La siguiente tabla muestra un array `a` en memoria de 5 elementos de tipo entero, que tiene sus valores inicializados al valor de su posición multiplicada por 10. En este caso, sus direcciones de memoria varían 4 bytes, ya que en esta arquitectura los valores enteros se almacenan ocupando 4 bytes. Como ya se ha expuesto, tenemos dos opciones para acceder a las direcciones de cada elemento: por un lado, es posible obtener la dirección utilizando el operador de dirección `&` junto al elemento del array (`&a[posición]`); por otro lado, utilizando directamente aritmética de punteros partiendo de la dirección inicial dada por el identificador del array (`a+posición`). De igual manera, podemos acceder a los valores del array utilizando indexación (`a[posición]`) o utilizando aritmética de punteros junto al operador de desreferencia `*` para obtener el valor apuntado por una dirección (`*(a+posición)`):

Array en memoria			Acceso a direcciones		Acceso a valores	
Dirección	**Posición**	**Valor**	**Indexación**	**Aritmética de punteros**	**Indexación**	**Aritmética de punteros**
000000b7e53ffc10	0	0	&a[0]	a	a[0]	*a
000000b7e53ffc14	1	10	&a[1]	a+1	a[1]	*(a+1)
000000b7e53ffc18	2	20	&a[2]	a+2	a[2]	*(a+2)
000000b7e53ffc1c	3	30	&a[3]	a+3	a[3]	*(a+3)
000000b7e53ffc20	4	40	&a[4]	a+4	a[4]	*(a+4)

En resumen, ahora tenemos tres opciones para acceder a los elementos de un array:

1. Utilizando la indexación de arrays.
2. Utilizando aritmética de punteros para calcular la dirección y acceder al elemento.
3. Utilizando un puntero que apunte directamente al elemento al que queremos acceder.

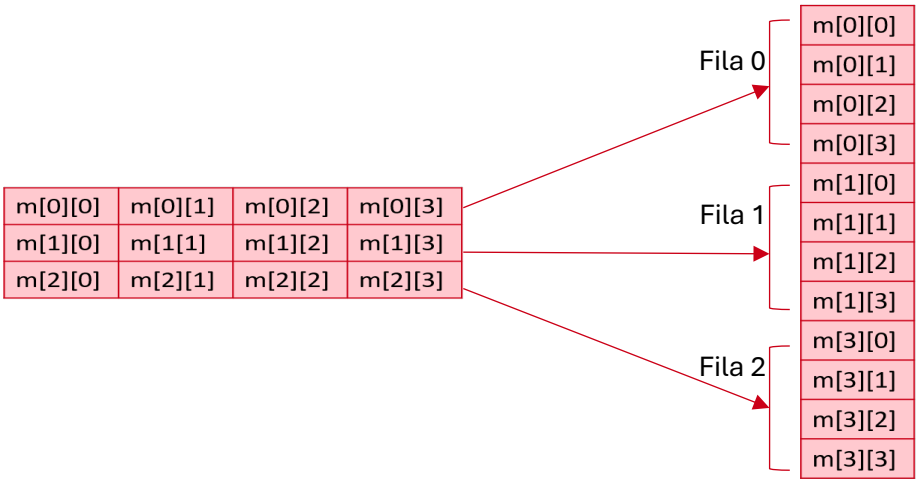
El siguiente ejemplo muestra diferentes formas de modificar el valor del elemento de la posición 2 del array:

```
int array[5];
// Acceso con indexación del array:
array[2] = 1;

// Acceso con aritmética de punteros para calcular la dirección:
*(array + 2) = 2;

// Acceso con un puntero auxiliar que apunta al elemento:
int *pArray = &array[2];
*pArray = 3;

// Similar a la anterior, pero iniciando en el primer elemento y
desplazando el número de posiciones:
pArray = array;
pArray += 2;
*pArray = 4;
```

Es necesario resaltar aquí que no es posible aumentar la dirección de memoria del propio array. Cogiendo como ejemplo el caso anterior, no podemos utilizar `array+=2` para apuntar al tercer elemento directamente con el identificador del array. Esto se debe a que, aunque los arrays puedan tratarse como punteros en determinadas ocasiones, estos son punteros inmutables, cuya dirección apuntada no puede variar.

10.2. Matrices

Como se especificó en el Tema 5, los elementos de una matriz o array bidimensional se almacenan en memoria de forma consecutiva por filas. Esto quiere decir que se almacenan en primer lugar los elementos de la primera fila, seguidos por los elementos de la segunda fila, etc. Por ejemplo, una matriz `m` de `3` filas y `4` columnas se almacenaría tal y como se muestra en la siguiente imagen:



De igual manera, también se ha detallado ya que el elemento situado en la posición `i`, `j` de una matriz `m` de dimensiones `NxM` puede accederse a través de índices de la forma: `m[i][j]`. Además, también es posible aprovechar el almacenamiento contiguo en memoria de sus elementos para acceder a ellos utilizando aritmética de punteros. En este

caso, es suficiente con un puntero que contenga la dirección del primer elemento (`m[0][0]`) y le sumemos el desplazamiento adecuado. Este desplazamiento se obtiene multiplicando el índice relativo a la fila (`i`) por el número de columnas en cada fila (`M`) y sumándole el índice correspondiente a la columna (`j`), quedando algo como:

```
int * pMatriz = &matriz[0][0];
*(pMatriz + i * M + j);
```

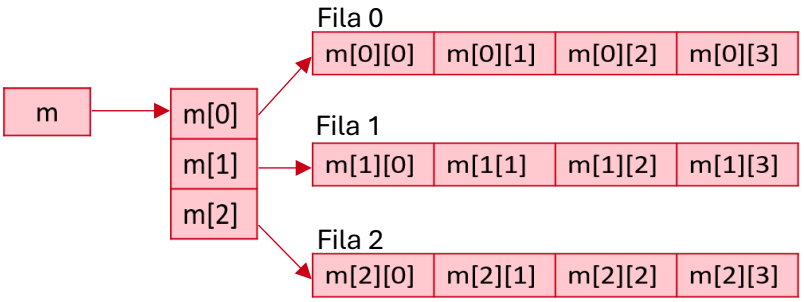
El siguiente ejemplo recorre una matriz inicializándola a los valores correspondientes a sus índices combinados, y después muestra por pantalla el elemento situado en la fila `1` y columna `2`:

```
#include <stdio.h>

#define N 3
#define M 4

int main() {
    int matriz[N][M];
    int * pMatriz = &matriz[0][0];
    for(int i = 0; i < N; i++)
    {
        for(int j = 0; j < M; j++)
        {
            *(pMatriz + (i * M) + j) = (i*M)+j;
        }
    }

    printf("%d\n", *(pMatriz + 1 * M + 2));

    return 0;
}
```

Esto también puede utilizarse para recorrer una matriz como si se tratase de un array unidimensional, como muestra el siguiente ejemplo, el cual es totalmente equivalente al ejemplo anterior:

```
#include <stdio.h>

#define N 3
#define M 4

int main() {
    int matriz[N][M];
    int * pMatriz = &matriz[0][0];

    for(int i = 0; i < N * M; i++)
    {
        *(pMatriz + i) = i;
    }

    printf("%d\n", *(pMatriz + 6));

    return 0;
}
```

Como puede apreciarse, en los dos ejemplos superiores se ha utilizado un puntero auxiliar que apunta al primer elemento de la matriz en lugar de utilizar el identificador de la matriz como puntero (de forma similar a como lo hacíamos con arrays unidimensionales). Esto no es posible en este caso, ya que el identificador de una matriz no es un puntero. Un array bidimensional también podía verse como un array de arrays, por lo que el identificador de un array bidimensional puede tratarse como un puntero a punteros, donde cada uno de estos punteros apunta al primer elemento de cada fila. Así, la matriz anterior `m` de `3` filas y `4` columnas también puede verse como:



Esto permite considerar a cada una de las filas como un array (o puntero a su primer elemento). Para ello, podemos utilizar el identificador del array bidimensional con una única dimensión. El siguiente ejemplo imprime el valor de los elementos de un array bidimensional utilizando los punteros relativos a cada fila para hacerlo:

```
int m[N][M] = {…};
int * fila;
for (int i = 0; i < N; i++) {
    fila = m[i];
    for (int j = 0; j < M; j++) {
        printf("%2d ", fila[j]);    // Acceso con indexación
    }
    printf("\n");
}
```

También es posible acceder a la fila y al elemento de la fila utilizando aritmética de punteros, como muestra el siguiente ejemplo equivalente al anterior:

```
int m[N][M] = {…};
int * fila;
for (int i = 0; i < N; i++) {
    fila = *(m + i); // Acceso con aritmética de punteros
    for (int j = 0; j < M; j++) {
        printf("%2d ", *(fila + j)); // Acceso con aritmética de
punteros
    }
    printf("\n");
}
```

De igual manera, podemos utilizar el propio identificador del array bidimensional como un puntero a puntero para acceder directamente al valor del elemento. En este caso es necesario utilizar dos operadores de desreferencia. El siguiente ejemplo recorre e imprime una matriz de esta forma:

```
int m[N][M] = {…};

for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        printf("%2d ", *(*(m + i) + j));  // Acceso con puntero a
puntero
    }
    printf("\n");
}
```

Estos tres últimos ejemplos son totalmente equivalentes. Finalmente, es necesario resaltar aquí que el identificador de un array bidimensional puede tratarse como un tipo especial de puntero a puntero, pero un puntero a puntero no tiene por qué ser un array bidimensional, por lo que estos conceptos no son intercambiables.

Tema 7: Funciones

En el lenguaje C, el uso de funciones es fundamental, ya que permite organizar el código de un programa en bloques reutilizables llamados subprogramas. Una función es un conjunto de instrucciones que realiza una tarea específica, y puede ser llamada desde cualquier parte del programa. Esto no solo promueve la modularidad, sino que también mejora la legibilidad y el mantenimiento del código. Además, las funciones también pueden recibir parámetros y devolver valores, permitiendo así una gran flexibilidad en la programación.

Un tipo especial de función es la recursiva, que se llama a sí misma para resolver problemas que pueden dividirse en subproblemas más pequeños. La recursión es una herramienta poderosa, particularmente útil en cálculos repetitivos como el cálculo de factoriales, o en algoritmos que requieren explorar estructuras de datos como árboles y grafos.

Hasta ahora, el código de los programas realizados se encontraba enteramente en la función `main`, haciendo nuestros programas poco modulares y escalables. Este tema se centra en la creación y uso de funciones en C, explorando su sintaxis, el paso de parámetros y el uso de la recursión, entre otros. Esto nos permitirá estructurar mejor nuestros programas, dividiendo las tareas en componentes manejables y eficientes.

1. Uso de funciones

Las funciones son bloques de código que se encargan de realizar una tarea específica dentro de un programa. Su uso es fundamental en la programación estructurada ya que:

- **Facilitan la reutilización del código**: evitan la duplicación de código al permitir que un conjunto de instrucciones se ejecute desde diferentes partes del programa sin tener que reescribirlas.
- **Mejoran la legibilidad y el mantenimiento**: dividir el programa en funciones más pequeñas y descriptivas hace que sea más sencillo de entender y depurar.
- **Permiten la modularidad**: cada función se puede ver como un "módulo" independiente que realiza una tarea específica.
- **Facilitan el trabajo en equipo**: al dividir un programa en funciones, diferentes programadores pueden trabajar en distintos módulos del programa simultáneamente.

A continuación, se muestra un ejemplo de una función denominada "`suma`", la cual encapsula la operación de sumar dos números, haciendo el código más organizado y claro:

```c
#include <stdio.h>

// Declaración de la función
int suma(int a, int b);

int main() {
    int num1 = 5, num2 = 3, resultado;
    // Llamada a la función
    resultado = suma(num1, num2);
```

©2024 Diego Hortelano Haro, Gerardo Reyes Salgado, Manuel Rubio Sánchez. Algunos derechos reservados. Este documento se distribuye bajo la licencia "Atribución/Reconocimiento-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en https://creativecommons.org/licenses/by-sa/4.0/deed.es.

```
    printf("La suma es: %d\n", resultado);
    return 0;
}

// Definición de la función
int suma(int a, int b) {
    return a + b;
}
```

2. Subprogramas y módulos

Con el fin de facilitar la legibilidad y el mantenimiento de un programa, el código suele dividirse en subprogramas y módulos:

- **Subprogramas**: son bloques de código, también conocidos como funciones o procedimientos, que realizan una tarea específica. En C, todos los subprogramas son funciones. Una función es invocada por el programa principal (o por otras funciones) y puede devolver un valor o no, dependiendo de su tipo de retorno.
- **Módulos**: un módulo es una parte del programa que agrupa subprogramas relacionados entre sí. La idea es dividir el programa en pequeños componentes, donde cada componente tiene una responsabilidad clara. El uso de módulos mejora la organización y el mantenimiento del código.

3. Diseño top-down

El diseño top-down es un enfoque de desarrollo de software en el cual se comienza diseñando el sistema en su nivel más alto de abstracción, dividiéndolo en módulos o funciones, y luego se descomponen estos en componentes más detallados. Este enfoque se basa en la idea de partir de lo general hacia lo específico. Este enfoque tiene las siguientes ventajas:

- Ayuda a tener una visión clara del sistema completo desde el inicio.
- Permite una planificación estructurada.
- Facilita la división de tareas y la asignación de funciones.

En resumen, la metodología top-down se basa en definir las funciones principales del programa e ir descomponiéndolas en funciones más pequeñas para tareas concretas.

4. Uso de funciones

Como ya se ha explicado, las funciones en C permiten estructurar un programa dividiéndolo en bloques lógicos. Cada función tiene una tarea específica que se puede invocar en cualquier parte del código y tantas veces como sea necesario, facilitando la reutilización de código y la simplificación de las tareas complejas. Sin embargo, aún no se ha detallado cómo se utilizan las funciones en C.

Para poder crear y utilizar nuestras funciones en el lenguaje C debemos seguir las siguientes etapas:

1. **Declaración**: se define la firma o prototipo de la función (tipo de retorno, identificador y parámetros).
2. **Definición**: se escribe el cuerpo de la función, especificando las instrucciones que ejecutará.
3. **Llamada**: se invoca la función desde el programa principal o desde otra función.

A continuación, se muestra un ejemplo que ilustra los tres pasos para trabajar con funciones, los cuales se detallan más abajo:

```c
#include <stdio.h>

// Declaración de la función
int cuadrado(int num);

// Cuerpo principal del código
int main() {
    int numero, resultado;
    // Solicitar al usuario un número
    printf("Introduce un número: ");
    scanf("%d", &numero);
    // Llamada a la función cuadrado
    resultado = cuadrado(numero);
    // Mostrar el resultado
    printf("El cuadrado de %d es: %d\n", numero, resultado);
    return 0;
}

// Definición de la función
int cuadrado(int num) {
    return num * num;  // Calcula el cuadrado del número
}
```

En el ejemplo destacan las tres etapas necesarias para la creación de nuestra propia función:

1. **Declaración**: al principio del código, antes de la función `main()`, se declara la función `cuadrado()` indicando el tipo del valor retornado (`int`), el identificador de la función (`cuadrado`) y el parámetro que recibe, junto a su tipo (`int num`). Es necesario resaltar aquí que el prototipo no incluye código, y termina con punto y coma `";"`:

   ```c
   int cuadrado(int num);
   ```

2. **Definición**: después de la función `main()`, se define la función `cuadrado()`, especificando lo que debe hacer: multiplicar el número recibido por sí mismo y retornar el resultado. El conjunto de instrucciones se encuentra en un bloque de código delimitado por llaves (`{}`):

   ```c
   int cuadrado(int num) {
       return num * num;  // Calcula el cuadrado del número
   }
   ```

3. **Llamada**: en el cuerpo de la función `main()`, se solicita al usuario un número, y luego se invoca a la función `cuadrado()`, pasando dicho número como argumento, para lo que se incluye la variable pasada como parámetro entre los paréntesis en la invocación de la función). Una vez se ejecute la función, su llamada se sustituirá por el valor devuelto, el cual se almacena en la variable `resultado` y se imprime por pantalla:

   ```c
   resultado = cuadrado(numero);
   ```

La ejecución del programa del ejemplo anterior mostraría:

```
Introduce un número:4
El cuadrado de 4 es: 16
```

Una función debe estar declarada antes de cualquier uso. Esto quiere decir que, al menos su prototipo debe aparecer en nuestro código antes de realizar cualquier invocación. Otra posibilidad es incluir directamente su definición antes de realizar cualquier invocación. Los siguientes apartados detallan cada uno de estos pasos.

4.1. Declaración de funciones

Como se ha expuesto antes, la declaración de una función en C se realiza antes de su uso en el código principal. Esta declaración consiste en especificar el tipo de retorno, el identificador de la función, y los parámetros que recibe, así como el tipo de cada parámetro. Esta declaración permite que el compilador reconozca la función, incluso si su definición aparece más adelante en el código. La declaración de una función (también denominada como prototipo de una función) tiene la siguiente sintaxis:

```
tipo_retorno identificador_funcion(tipo_parametro1, tipo_parametro2, ...);
```

Donde:

- **`tipo_retorno`**: el tipo de dato que la función devuelve (`int`, `float`, `char`, etc.). Si la función no retorna ningún valor, se usa `void`.
- **`identificador_funcion`**: el nombre que identifica a la función.
- **`tipo_parametro1`**, **`tipo_parametro2`**: los tipos de los parámetros que recibe la función. Si la función no recibe ningún parámetro los paréntesis deben dejarse vacíos. Además de los tipos, es posible indicar el identificador de los parámetros, de manera similar a como lo hacemos en la definición de la función.

A continuación, se muestra un ejemplo de la declaración de una función:

```
int suma(int a, int b);
```

4.2. Llamada a funciones

Para llamar o invocar a una función en C, se utiliza su identificador, seguido de paréntesis con los argumentos correspondientes (si los tiene). La llamada puede realizarse desde el programa principal o desde otra función. Su sintaxis es:

```
identificador_funcion(parametro1, parametro2, ...);
```

Donde:

- **`identificador_funcion`**: el nombre que identifica a la función que deseamos invocar.
- **`parametro1`**, **`parametro2`**, **`...`**: son los argumentos pasados como parámetro a la función. Los tipos de los datos incluidos en la llamada de una función deben coincidir con los tipos especificados en la declaración y definición de dicha función. Si la función no requiere ningún parámetro, bastará con incluir los paréntesis vacíos. Estos parámetros reciben el nombre de parámetros actuales.

La ejecución de la función reemplaza la llamada por el valor que retorne, en caso de que sea una función que devuelve algo. A continuación, se muestra un ejemplo de la llamada a una función desde la función `main()`:

```
int main() {
    int x = 4, y = 5;
    int resultado = sumar(x, y);  // Llamada a la función
    printf("El resultado es: %d\n", resultado);
    return 0;
}
```

En este ejemplo, se invoca a la función `suma`, la cual se ejecutará y retornará un valor de tipo `int`, que se almacenará en la variable `resultado`. Es necesario resaltar aquí que la invocación de una función siempre incluirá los paréntesis, aunque la función no requiera parámetros y estos vayan vacíos.

4.3. Definición de una función

La definición de una función es el proceso de escribir el cuerpo de la función, detallando lo que hace cuando es invocada. Dicho de otra forma, la definición de una función es la escritura de la lógica o cuerpo de la función. Su sintaxis es:

```
tipo_retorno id_funcion(tipo_par1 nombre_par1, tipo_par2 nombre_par2, ...) {
 // Cuerpo de la función
}
```

Donde:

- **`tipo_retorno`**: es el tipo de dato que devuelve o retorna la función cuando finalice su ejecución (por ejemplo, `int`, `void`).
- **`id_funcion`**: el identificador o nombre de la función.
- **`tipo_par1`, `tipo_par2`**: el tipo de los parámetros que recibe la función, si los tiene.
- **`nombre_par1`, `nombre_par2`**: indican los nombres de los parámetros, si los hay. Estos parámetros se denominan parámetros formales.

A continuación, se incluye un ejemplo de la definición de una función suma, la cual requiere dos parámetros de tipo `int` y retorna un valor, también de tipo `int`:

```
int sumar(int a, int b) {
    return a + b;
}
```

Es necesario resaltar aquí la palabra clave `return`. Esta sentencia se utiliza para finalizar la ejecución de una función, pudiendo devolver un valor a la invocación de la función. Resulta de vital importancia para controlar el flujo de ejecución de las funciones y para comunicar resultados a otras partes del programa. Sin embargo, la sentencia `return` puede retornar únicamente un valor, por lo que necesitaremos utilizar otros mecanismos, como el paso de parámetros por referencia, si necesitamos devolver más resultados.

5. Ámbito de una variable

El ámbito en C se refiere a la visibilidad y vida útil de una variable dentro del programa. En general, hay tres tipos de ámbito:

1. **Ámbito global**: las variables declaradas fuera de cualquier función son accesibles en todo el programa. Se recomienda evitar su uso porque dificultan la reutilización, modulación y mantenimiento de nuestro programa.
2. **Ámbito local**: las variables declaradas dentro de una función solo son accesibles dentro de esa función.
3. **Ámbito de bloque**: las variables definidas dentro de un bloque (delimitado con `{}`) únicamente existen en dicho bloque. Es común utilizar este ámbito en instrucciones de selección y repetición.

A continuación, se muestra un ejemplo con una variable de ámbito global denominada "`global`" y otra de ámbito local, denominada "`local`". En este caso la variable global puede ser accedida desde cualquier función, mientras que la variable local solo puede ser utilizada dentro de la función `main()`:

```c
#include <stdio.h>

int global = 10;  // Variable de ámbito global

void imprimirGlobal() {
    printf("Valor de la variable global: %d\n", global);
}

int main() {
    int local = 5;  // Variable de ámbito local
    imprimirGlobal();
    printf("Valor de la variable local: %d\n", local);
    return 0;
}
```

6. Ejecución de funciones

Cuando una función se invoca en C, el control del programa se transfiere a la función que se ejecuta de forma secuencial. Al finalizar, el control regresa al punto donde fue invocada la función, retornando el valor si la función no es de tipo `void`, en cuyo caso no retorna nada. De esta forma, el flujo de ejecución es el siguiente:

1. El programa principal o la función que realiza la invocación transfiere el control a la función invocada.
2. La función recibe una copia del valor de los parámetros actuales (los parámetros que se pasan a la misma), los cuales se copian en los parámetros formales.
3. La función ejecuta sus instrucciones.
4. Si la función tiene un valor de retorno, lo devuelve al punto de llamada para que sea almacenado en una variable o procesado en una expresión.
5. El programa principal retoma el control del programa y continúa con la ejecución de instrucciones.

7. Paso de parámetros

El paso de parámetros es el mecanismo mediante el cual se transmiten datos a una función para que ésta los utilice durante su ejecución. El paso de parámetros a las funciones puede realizarse por valor o por referencia.

- **Paso de parámetros por valor**: cuando una función recibe un parámetro por valor, se le pasa una copia del valor original. Esto significa que los cambios realizados dentro de la función no afectan al valor original. El paso de parámetros por valor es especialmente útil cuando solo necesitamos el valor del parámetro, pero no necesitamos modificar el valor del original. A continuación, se muestra un ejemplo de paso de parámetros por valor:

```c
#include <stdio.h>

void cambiarValor(int x) {
    x = 10;  // Cambia el valor de la copia local, no el
original
}

int main() {
    int numero = 5;
    cambiarValor(numero);
    printf("Valor de numero: %d\n", numero);  // Imprime 5
    return 0;
}
```

  En el ejemplo anterior, la función modifica el valor de la variable `x`, el parámetro formal de la función `cambiarValor()` que es de ámbito local, sin modificar el valor original de la variable `numero`, pasada como parámetro actual en la llamada a la función.

- **Paso de parámetros por referencia**: este tipo de paso de parámetros proporciona acceso a la variable original, permitiendo modificar su valor. En C, el paso de parámetros siempre se realiza como una copia del valor pasado. Por ello, si queremos modificar la variable original es necesario pasar como parámetro una referencia o puntero a la dirección de memoria de variable original. Así, aunque se realice una copia del valor pasado como parámetro, podremos utilizar dicha dirección de memoria para acceder al valor original, lo que permite que los cambios realizados dentro de la función afecten al valor original. Este tipo de paso de parámetros, además de permitir modificar el valor original de una variable, permite ahorrar memoria si trabajamos con datos grandes (no se copian todos los datos, solo una dirección de memoria. A continuación, se muestra un ejemplo de paso de parámetros por referencia:

```c
#include <stdio.h>

void cambiarValorPorReferencia(int *x) {
    *x = 10;  // Cambia el valor original a través del puntero
}

int main() {
    int numero = 5;
    cambiarValorPorReferencia(&numero);
```

```
    printf("Valor de numero: %d\n", numero);  // Imprime 10
    return 0;
}
```

En este caso, se realiza una copia de la dirección de memoria de la variable número, almacenando dicha dirección en el puntero x al invocar a la función. Este puntero se utiliza para acceder a la dirección de memoria de la variable original y modificar su contenido.

8. Paso de arrays como parámetro

En C, los arrays son punteros que apuntan a su primer elemento. Por ello, siempre se pasan a las funciones por referencia. Esto significa que cualquier modificación que se haga al array dentro de la función afectará al array original, ya que se pasa la dirección base del array.

Además, en la función no es posible conocer el tamaño del array, y, si utilizamos el operador `sizeof()`, este nos dará el tamaño de un puntero, no del tamaño del array. Así, la única opción para poder trabajar con un array en una función es pasar como parámetro también su longitud. A continuación, se muestra un ejemplo de una función que recibe como parámetro un array (el puntero a su primer elemento) y su tamaño (el número de elementos que podemos recorrer). Además, al utilizar el paso por referencia, los valores del array inicial se modificarán:

```
#include <stdio.h>

void modificarArray(int * arr, int tamaño) {
    for (int i = 0; i < tamaño; i++) {
        arr[i] *= 2;  // Duplica cada valor del array
    }
}

int main() {
    int numeros[] = {1, 2, 3, 4, 5};
    int tamaño = 5;
    modificarArray(numeros, tamaño);
    // Imprime el array modificado
    for (int i = 0; i < tamaño; i++) {
        printf("%d ", numeros[i]);  // Imprime 2 4 6 8 10
    }
    return 0;
}
```

9. Recursividad

La recursividad es una técnica en la que una función se llama a sí misma directa o indirectamente, resolviendo un problema dividiéndolo en subproblemas más pequeños. La recursividad se utiliza para resolver problemas que tienen una naturaleza repetitiva, como calcular factoriales o resolver problemas de recorridos o algoritmos de búsqueda.

Al utilizar la recursividad debemos pensar de manera declarativa: nos interesa pensar en qué vamos a hacer en lugar de en cómo vamos a hacerlo. Para ello suponemos que conocemos qué se resuelve (un subproblema de nuestro problema inicial) en lugar de cómo se resuelve. Además, es necesario utilizar la función recursiva que estamos

implementando, asumiendo que funciona, aunque aún no hayamos terminado de implementarla. Al utilizar recursividad aparecen dos conceptos clave:

- Descomposición o simplificación de problemas. Ciertos problemas pueden resolverse utilizando soluciones a problemas idénticos al original, pero más sencillos o de menor tamaño.
- Inducción. Nuestra solución debe suponer que ya sabemos la solución a los problemas más simples. Las subsoluciones o soluciones parciales que nos permitirán llegar a la solución del problema inicial se obtendrán a través de llamadas recursivas.

A la hora de diseñar algoritmos recursivos se sigue la siguiente metodología:

1. Identificación del **tamaño del problema**, que determinará el número de operaciones a realizar.
2. Establecimiento de los **casos base**. Los casos base son instancias sencillas de nuestro problema, que se resuelven directamente, sin necesidad de llamadas recursivas. Esto permite interrumpir la sucesión de llamadas a la propia función, permitiendo su finalización.
3. **Descomposición**. Selección de varias instancias del problema inicial de menor tamaño.
4. **Inducción**. Establecimiento de los casos recursivos a partir de las soluciones de las instancias seleccionadas antes.
5. **Implementación** y **pruebas** de nuestra función recursiva.

A continuación, se muestra un ejemplo que sigue esta metodología para implementar una solución recursiva al problema del cálculo del factorial:

$$F(n) = n! = \prod_{i=1}^{n} i = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n$$

1. Identificación del **tamaño del problema**: en este caso, necesitamos multiplicar todos los números enteros desde n hasta 1, por lo que tenemos un tamaño lineal en n. Esto significa que el número de operaciones crece proporcionalmente con n.
2. Establecimiento de los **casos base**: las instancias más sencillas de este problema se corresponderán con el cálculo del factorial de números bajos. Por ejemplo, podemos considerar:

   a. $F(2) = 2! = 1 \times 2 = 2$
   b. $F(1) = 1! = 1$
   c. $F(0) = 1$ (por conveniencia mátemática)

   En este caso, el caso base $F(1)$ contiene al caso $F(2)$, por lo que es un candidato muy bueno a ser un caso base. Si consideramos el caso $F(0)$, también podemos utilizarlo como caso base.

3. **Descomposición**: es necesario considerar diferentes opciones para obtener problemas de menor tamaño. Algunos de estos problemas suelen ser: $n-1, n-2$, $n/2, n/10$... En este caso, el subproblema inmediatamente anterior al cálculo del

factorial de un número es el cálculo del factorial del número anterior, por lo el subproblema $n-1$ puede ser una muy buena opción.

4. **Inducción**: debemos ser capaces de hallar la solución a nuestro problema suponiendo que tenemos la solución del subproblema elegido. En caso de no ver un patrón claro, podemos volver al paso anterior y elegir otro subproblema. Para el caso de n y $n-1$ podemos plantear los siguientes supuestos:

    a. Si el problema es 3 ($n=3$), el subproblema $n-1$ sería 2. Conociendo que $2!=2$, ¿puedo obtener $3!$? En este caso podemos ver de manera sencilla que $3!=6=1\times2\times3=2!\times3$.
    b. Si el problema es 5 ($n=5$), el subproblema $n-1$ sería 4. Conociendo que $4!=24$, ¿puedo obtener $5!$? De nuevo en este caso podemos ver de manera sencilla que $5!=120=1\times2\times3\times4\times5=4!\times5$.
    c. Por tanto, de manera generalizada, podemos suponer que el factorial de un número n viene dado por el factorial del número anterior $(n-1)$ multiplicado por el propio número n, o como ecuación: $n!=(n-1)!\times n$.

5. Utilizando la inducción y el caso base propuestos podemos **implementar** nuestra función recursiva para el cálculo del factorial, quedando:

```c
// Función recursiva para calcular el factorial de un número
#include <stdio.h>

int factorial(int n) {
    if (n == 0) {
        return 1; // Caso base o condición de parada
    } else {
        return n * factorial(n - 1); // Llamada recursiva
    }
}

int main() {
    int num = 5;
    printf("El factorial de %d es: %d\n", num, factorial(num)); // Imprime 120
    return 0;
}
```

En este ejemplo, la función `factorial` se invoca a sí misma hasta que alcanza el caso base (`n==0`), en el cual la función retorna 1 y las llamadas recursivas comienzan a resolverse, calculando el resultado del problema inicial.

10.Función main

La función `main` es el punto de entrada de todo programa en C. Es donde comienza la ejecución del programa, y su declaración es obligatoria en cualquier programa escrito en este lenguaje. Generalmente, la función `main` devuelve un valor entero (`int`), que usualmente se usa para indicar si el programa finalizó correctamente. La sintaxis de la función `main` es:

```c
int main() {
    // Cuerpo del programa
    return 0;  // Indica que el programa terminó correctamente
}
```

El retorno en el caso de la función `main()` suele ser:

- Return 0: indica una terminación correcta del programa.
- Return diferente de 0: puede indicar un error u otro estado.

Además, hay otra definición de la función main(), la cual admite dos parámetros genéricos, los cuales son:

- `argc` (argument count): número de parámetros o argumentos que ha recibido la función `main()`.
- `argv` (argument vector): vector o array con los parámetros recibidos por la función. Este parámetro es un array de punteros a cadenas de caracteres, por lo que puede contener tantas cadenas de caracteres como sean necesarias (el número lo indica el parámetro `argc`).

Estos argumentos se reciben de la línea de comandos (si ejecutamos nuestro programa desde utilizando la consola) o desde la configuración del IDE. Además, siempre hay al menos un parámetro, y es que la primera posición del array (`argv[0]`) se corresponde con el nombre de nuestro programa.

A continuación, se muestra un ejemplo de acceso a los parámetros de la función `main()`:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Número de argumentos: %d\n", argc);
    for (int i = 0; i < argc; i++) {
        printf("Argumento %d: %s\n", i, argv[i]);
    }
    return 0;
}
```

Es necesario resaltar aquí que las posiciones del array `argv` son cadenas de caracteres, por lo que deberemos pasarlas de manera correcta a otros tipos de datos para poder procesarlas.

11.Módulos

Los módulos en programación se refieren a la división del programa en varias partes o bloques que se pueden gestionar de forma independiente. En C, esta modularidad se logra mediante el uso de archivos de código fuente (*.c*) y archivos de cabecera (*.h*):

1. **Archivo de cabecera (.h)**: contiene las declaraciones de las funciones y variables globales que se van a compartir entre varios archivos de código.
2. **Archivo de código fuente (.c)**: contiene las definiciones y la implementación de las funciones y variables.

El uso de módulos facilita el mantenimiento del código, la reutilización de funciones en distintos programas y la separación lógica de las partes del código.

A continuación, se muestra un ejemplo de un archivo de cabecera denominado "`calculadora.h`". Dicho archivo contiene las declaraciones (prototipos) de dos funciones. Además, utiliza las directivas `#ifndef` para evitar múltiples copias si se intenta incluir múltiples veces el módulo:

```
#ifndef CALCULADORA_H_
#define CALCULADORA_H_

int sumar(int a, int b);
int restar(int a, int b);

#endif /* CALCULADORA_H_ */
```

Por otra parte, en el mismo proyecto tendremos también el fichero "`calculadora.c`", que contendrá la declaración de las funciones, es decir su cuerpo. Este fichero debe incluir la su cabecera correspondiente, como muestra el siguiente ejemplo:

```
#include "calculadora.h"

int sumar(int a, int b) {
   return a+b;
}

int restar(int a, int b) {
   return a-b;
}
```

Finalmente, es necesario incluir el módulo en nuestro programa para hacer uso de sus funciones. Para ello se utiliza la directiva `#include`, pero con comillas dobles (" ") en lugar de los corchetes angulares (<>), al estar el módulo en nuestro proyecto:

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

int main() {
   int num1 = 10, num2 = 5;
   printf("Suma: %d\n", sumar(num1, num2));  // Imprime 15
   printf("Resta: %d\n", restar(num1, num2));  // Imprime 5
   return 0;
}
```

Para poder compilar y ejecutar sin problemas nuestro código, será necesario añadir el módulo creado al código a compilar. Para ello, en CLion es suficiente con incluirlo en la sentencia `add_executable()` del fichero `CMakeList.txt` de nuestro proyecto:

```
add_executable(ejemplos main.c calculadora.h calculadora.c)
```

Tema 8: Memoria Dinámica

La memoria en C se divide en diferentes bloques según su propósito y tiempo de vida. Es necesario comprender el uso las características de cada uno de estos bloques para gestionar la memoria de manera eficiente y evitar errores. Hasta ahora, hemos estado utilizando algunos de bloques sin ser conscientes de ello. Este tema presenta estos bloques en detalle, centrándose especialmente en el montículo o heap, utilizado para la asignación dinámica de memoria.

1. Espacios de memoria en C

La memoria en C se divide en cuatro bloques diferentes de memoria, en función de su cometido y tiempo de vida. Estos bloques son:

1. **Segmento de Código o Texto**:
    - Contiene el código (instrucciones) del programa que se está ejecutando.
    - Este bloque de memoria es de solo lectura y tiene un tamaño fijo durante toda la ejecución del programa.
2. **Segmento de Datos**:
    - Contiene variables globales y estáticas (declaradas con la palabra clave static), y se divide en dos:
        i. Datos inicializados, que incluye las variables globales y estáticas que se inicializan con un determinado valor antes de la ejecución del programa.
        ii. Datos sin inicializar (o BSS, Block Started by Symbol), que contiene las variables globales y estáticas sin inicializar, las cuales se inicializan con ceros automáticamente.
3. **Pila (o Stack):**
    - Almacena las variables locales de las funciones, así como la información de las llamadas a funciones (parámetros y direcciones de retorno).
    - Las variables almacenadas en este bloque de memoria tienen un tiempo de vida limitado ligado a la función donde están declaradas. Cuando la función termina, el espacio de memoria se libera automáticamente.
    - Este bloque de memoria recibe este nombre porque las funciones se "apilan", ya que, si una función invoca a otra, la primera queda bloqueada hasta que la llamada a la segunda finaliza. Esto quiere decir que la memoria reservada por la primera función no podrá liberarse hasta que la memoria reservada por la segunda función se libere al finalizar su ejecución.
    - En este caso, las variables no inicializadas adquieren el valor que haya en memoria en el momento en que se reserva.
    - La principal ventaja de este bloque de memoria es su rápido acceso, pero tiene un tamaño limitado que depende del sistema utilizado.
4. **Montículo (o Heap):**
    - Se utiliza para la asignación dinámica de memoria en tiempo de ejecución.

©2024 Diego Hortelano Haro, Gerardo Reyes Salgado, Manuel Rubio Sánchez. Algunos derechos reservados. Este documento se distribuye bajo la licencia "Atribución/Reconocimiento-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en https://creativecommons.org/licenses/by-sa/4.0/deed.es.

- Este bloque no tiene un tamaño fijo, y no se reserva ni se libera automáticamente, siendo el programador el encargado de realizar estas acciones.
- Por ello, el tiempo de vida de las variables en este bloque de memoria es flexible, permitiendo conservar datos mientras lo necesitemos.

Debemos tener cuidado al utilizar los diferentes bloques de memoria, ya que es posible provocar errores relacionados con ellos. Los errores más comunes son:

- Desbordamiento de la pila, si utilizamos demasiada memoria, por ejemplo, con llamadas recursivas demasiado profundas (cada llamada reservará de nuevo espacio para la ejecución de esa función).
- Fugas de memoria, si olvidamos liberar la memoria del montículo, consumiendo espacio que no se recupera.
- Acceso inválido, si intentamos acceder a variables que ya no existen o no están reservadas, como variables fuera de su ámbito o punteros que apuntan a una dirección no válida o que ya se ha liberado.

2. Memoria dinámica

La memoria dinámica es aquella que hace uso del bloque denominado montículo o heap. Este bloque permite la reserva y liberación de memoria durante la ejecución de nuestro programa, lo que la hace especialmente interesante cuando no sepamos cuantos datos necesitamos almacenar en el tiempo de compilación.

Si utilizamos memoria estática (por ejemplo, con variables locales), tenemos dos opciones: bien reservamos el tamaño máximo que podríamos necesitar, lo que supone un sobredimensionamiento en la mayoría de los casos y un desperdicio de memoria; o bien utilizamos arrays de longitud variable (VLA), que permiten su definición en tiempo de ejecución, aunque no es posible utilizarlos en todos los estándares de C. La opción más recomendada en este caso es el uso de memoria dinámica.

La siguiente tabla recoge las principales diferencias entre memoria estática y memoria dinámica:

	Memoria estática	**Memoria dinámica**
Almacenamiento	Pila (stack)	Montículo (heap)
Variables	Disponibles en función de su ámbito	Disponibles durante toda la ejecución (hasta su liberación)
Gestión de memoria	Automática	Reserva y liberación de espacio realizadas por el programador
Asignación de memoria	Más rápida	Más flexible, incluso pudiendo redefinir espacios

La gestión de la memoria dinámica en C se realiza a través de punteros, ya que nos permiten referenciar zonas de memoria (en este caso, de memoria dinámica), y llamadas al sistema que nos permiten reservar y liberar zonas de memoria. Las principales funciones para la gestión de memoria se detallan en el siguiente punto.

3. Funciones para la gestión de memoria

Como se ha comentado en los puntos anteriores, la memoria dinámica es aquella que se reserva en el montículo o heap en tiempo de ejecución. Estas reservas producen variables dinámicas, que son posiciones de memoria que se reservan dinámicamente.

La forma de utilizar estas variables es a través de su dirección de memoria, por lo que es imprescindible el uso de punteros. Además, la gestión de memoria (reserva, reubicación y liberación) se realiza con llamadas al sistema a través de funciones de la biblioteca estándar `<stdlib.h>`, las cuales aceptan o devuelven (o ambas) valores de tipo puntero. Al no conocer el tipo de los datos con los que vamos a trabajar, éstos serán punteros genéricos (`void`).

Las principales funciones para la gestión de la memoria dinámica son `malloc()`, `calloc()`, `realloc()` y `free()`, y se detallan en los siguientes puntos.

3.1. Función malloc()

La función `malloc()` (memory allocation) reserva un bloque **contiguo** de memoria del tamaño pasado como parámetro, lo que permite reservar memoria tanto para variables como para arrays. La sintaxis de esta función es:

```
void * malloc (int num_bytes)
```

Donde:

- `num_bytes` es el número de bytes que deseamos reservar. Usualmente suele combinarse con el operador `sizeof()`, para reservar espacio para el número de elementos del tipo deseado.

Esta función retorna un puntero genérico que apunta a la dirección de inicio del bloque de memoria reservado si se ha podido realizar la reserva. En caso contrario, la función retornará `NULL`. Por ello, es necesario realizar un casting al tipo de datos que queremos reservar, y se considera buena práctica comprobar que la dirección retornada es válida.

El siguiente ejemplo muestra la reserva de una variable (un único elemento) de tipo entero y de un array de 10 elementos en punto flotante utilizando punteros (`pI` y `pF`) y la función `malloc()`. El uso de estas variables se realiza de manera similar a las variables vistas hasta ahora. Incluso es posible recorrer un array utilizando indexación de arrays:

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

int main() {
   int num_elementos = 10;
   int * pI;  // Puntero a entero
   float * pF; // Puntero a float
   pI = (int *) malloc(sizeof(int));  // Reservamos memoria para un único entero
   if(pI == NULL)  // Comprobamos que la posición de memoria es válida
   {
      printf("No se ha podido reservar memoria para un entero. Finalizando programa.\n");
      return -1;
   }
```

```
    pF = (float *) malloc(num_elementos * sizeof(float)); // Reservamos memoria para un array de 10 elementos de tipo float
    if(pF == NULL)  // Comprobamos que la posición de memoria es válida
    {
        printf("No se ha podido reservar memoria para el array de floats. Finalizando programa.\n");
        return -2;
    }

    // Uso de las variables reservadas utilizando punteros:
    *pI = 20;

    for(int i = 0; i < num_elementos; i++)
    {
        *(pF + i) = i * 0.2f;  // Inicializamos cada posición a un valor float diferente
    }

    // También es posible recorrer la memoria reservada utilizando indexación de arrays:
    for(int i = 0; i < num_elementos; i++)
    {
        printf("%f\t", pF[i]);  // Mostramos los diferentes elementos
    }

    return 0;
}
```

3.2. Función calloc()

La función `calloc()` (clear allocation) también reserva un bloque contiguo de memoria en el montículo o heap, de manera similar a como lo hace la función `malloc()`. Las principales diferencias con la función anterior es que `calloc()` inicializa el espacio reservado con ceros, y sus parámetros son ligeramente diferentes. Su sintaxis es:

```
void * calloc (int num_elementos, int tamaño_elemento)
```

Donde:

- `num_elementos` es el número de elementos que queremos reservar en memoria.
- `tamaño_elemento` es el tamaño de cada uno de los elementos reservados.

Así, esta función reserva un espacio total de `num_elementos * tamaño_elemento`.

Al igual que la función `malloc()`, `calloc()` retorna un puntero genérico (por lo que debemos realizar un casting) a la primera dirección del bloque de memoria reservado, o `NULL` si no se ha podido realizar la reserva (por lo que debemos comprobar que la reserva se ha realizado correctamente).

El siguiente ejemplo, similar al anterior, muestra el uso de la función `calloc()`. Este ejemplo también se utiliza la función `calloc()` para reservar memoria dinámica en el montículo o heap para una variable entera y un array de 10 elementos, que en este caso se inicializarán a 0. El uso de estas variables se realiza de la misma manera que antes.

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

int main() {
   int num_elementos = 10;
   int * pI;   // Puntero a entero
   float * pF; // Puntero a float
   pI = (int *) calloc(1, sizeof(int));   // Reservamos memoria para un único entero
   if(pI == NULL)  // Comprobamos que la memoria es válida
   {
      printf("No se ha podido reservar memoria para un entero. Finalizando programa.\n");
      return -1;
   }

   pF = (float *) calloc(num_elementos, sizeof(float)); // Reservamos memoria para un array de 10
elementos de tipo float
   if(pF == NULL)// Comprobamos que la memoria es válida
   {
      printf("No se ha podido reservar memoria para el array de floats. Finalizando programa.\n");
      return -2;
   }

   // Uso de las variables reservadas utilizando punteros:
   *pI = 20;

   // Mostramos el contenido del array (debe estar inicializado a ceros):
   for(int i = 0; i < num_elementos; i++)
   {
      printf("%f\t", pF[i]);  // Mostramos los diferentes elementos
   }

   return 0;
}
```

3.3. Función realloc()

La función `realloc()` (re-allocation) permite redimensionar el tamaño de un bloque de memoria que ha sido previamente reservado con `malloc()` o `calloc()`. Su sintaxis es la siguiente:

```
void * realloc (void * puntero_anterior, int num_bytes)
```

Donde:

- `puntero_anterior` es la dirección de memoria inicial del bloque de memoria previamente reservado.
- `num_bytes` es el nuevo tamaño (en bytes) que tendrá el bloque.

El funcionamiento de esta función puede variar según el tamaño del nuevo bloque:

- Si el nuevo tamaño es mayor que el tamaño actual del bloque de memoria, y la memoria contigua al bloque actual no está reservada, `realloc()` expande el bloque de memoria y preserva los valores existentes. En este caso, el puntero retornado tiene la misma dirección que el puntero anterior.

- Si el nuevo tamaño es mayor que el tamaño actual del bloque y no es posible expandirlo, la función `realloc()` asigna un nuevo bloque de memoria, copia los datos del bloque anterior al nuevo bloque y libera la memoria del bloque original. En este caso, la función puede retornar `NULL` si no hay espacio suficiente para asignar el nuevo tamaño.
- Si el nuevo tamaño es menor, se reduce el bloque actual, conservando los valores hasta el nuevo límite. En este caso, el puntero retornado apuntará al mismo bloque que el puntero anterior.
- Si el nuevo tamaño es 0, `realloc()` actúa como la función `free()` que se detalla en el siguiente apartado, y libera el bloque de memoria anterior. En este caso, la función retornara `NULL`.

Así, al igual que las funciones anteriores, `realloc()` retorna un puntero genérico que puede contener la misma dirección que la pasada como parámetro, una nueva dirección que apunte a un nuevo bloque o ser `NULL`. Por ello, al igual que en los casos anteriores, es necesario realizar un casting y comprobar que dicho puntero tiene una dirección válida.

El siguiente ejemplo muestra el uso de la función `realloc()`. Este ejemplo utiliza en entero al que asigna una posición en memoria dinámica para un único entero, al que le asigna el valor 20. Después, utiliza la función `realloc()` para expandir el espacio a 10 (`num_elementos`) elementos, imprimiendo todos ellos. Todos estos elementos estarán sin inicializar, excepto el primer elemento, ya que asignamos el valor 20 a la primera posición antes de redimensionar el bloque:

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

int main() {
   int num_elementos = 10;
   int * pI;   // Puntero a entero
   pI = (int *) calloc(1, sizeof(int));   // Reservamos memoria para un único entero
   if(pI == NULL) { // Comprobamos que la memoria es válida
      printf("No se ha podido reservar memoria para un entero. Finalizando programa.\n");
      return -1;
   }
   *pI = 20;  // Acceso a la variable dinámica reservada:

   // Redimensionamos el bloque de memoria para que pueda almacenar num_elementos enteros:
   pI = (int *) realloc(pI, num_elementos * sizeof(int));

   if(pI == NULL){  // Comprobamos que el nuevo bloque de memoria es válido
      printf("No se ha podido reservar memoria para un entero. Finalizando programa.\n");
      return -1;
   }

   // Mostramos el contenido del array (el primer valor debe ser 20, el valor asignado anteriormente):
   for(int i = 0; i < num_elementos; i++){
      printf("%f\t", pI[i]);  // Mostramos los diferentes elementos
   }

   return 0;
}
```

3.4. Función free()

La función `free()` libera el bloque de memoria dinámica reservado previamente con `malloc()` o `calloc()` (o redimensionado con `realloc()`). Su sintaxis es la siguiente:

```
void free (void * puntero_bloque)
```

Donde:

- `puntero_bloque` es la dirección de memoria inicial del bloque de memoria a liberar.

Al utilizar memoria dinámica es responsabilidad del programador reservar la memoria que necesita. Asimismo, cualquier bloque de memoria reservado dinámicamente debe ser liberado en algún momento; esta tarea también recae en el programador, quien debe hacerlo mediante la función `free()`. Si no liberamos la memoria después de su uso, el programa retiene la memoria reservada, lo que puede llevar al agotamiento de la memoria disponible o a la reducción del rendimiento del sistema.

Los principales errores que podemos cometer relacionados con esta función son:

- Invocar a la función `free()` con un puntero que no apunta a memoria asignada con las funciones `malloc()`, `calloc()` o `realloc()`, lo que provoca un comportamiento indefinido y un posible fallo del sistema.
- Invocar a la función `free()` más de una vez con el mismo puntero sin reasignarlo, lo que puede provocar el fallo del programa debido a la corrupción de memoria.
- Utilizar el puntero para acceder a la memoria después de haberla liberado con `free()` puede provocar errores de segmentación o acceso a datos incorrectos.

Es importante tener en cuenta que la función `free()` no cambia el valor del puntero pasado como parámetro, simplemente marca la memoria como disponible para ser reutilizada. Esto quiere decir que el puntero aún contiene la dirección de la memoria liberada, lo que se conoce como puntero colgante (dangling pointer). Por ello, es una buena práctica establecer el puntero a `NULL` después de liberar la memoria para evitar accesos accidentales a un bloque de memoria que ya no está reservado.

El siguiente ejemplo muestra el uso de la función `free()`. En el ejemplo se reserva memoria para una variable dinámica de tipo entero. Tras utilizar dicha variable, se hace uso de la función `free()` para liberar la memoria reservada, asignando al puntero el valor `NULL`. Después, se reutiliza el mismo puntero para almacenar la dirección de un bloque de memoria de 10 enteros. Una vez utilizado este nuevo bloque, volvemos a liberarlo con `free()`, colocando de nuevo el puntero a `NULL`:

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

int main() {
   int num_elementos = 10;
   int * pI;   // Puntero a entero

   pI = (int *) malloc(sizeof(int));   // Reservamos memoria para un único entero
   if(pI == NULL)  // Comprobamos que la memoria es válida
```

```
    {
        printf("No se ha podido reservar memoria para un entero. Finalizando programa.\n");
        return -1;
    }

    // Aquí podríamos utilizar el puntero pI, que apunta a una variable dinámica de tipo int

    // Liberamos memoria y ponemos el puntero a NULL:
    free(pI);
    pI = NULL;

    // Reservamos un nuevo bloque de memoria para que pueda almacenar num_elementos enteros:
    pI = (int *) malloc(num_elementos * sizeof(int));
    if(pI == NULL)  // Comprobamos que el nuevo bloque de memoria es válido
    {
        printf("No se ha podido reservar memoria para un entero. Finalizando programa.\n");
        return -1;
    }

    // Aquí podríamos utilizar el puntero pI con un bloque de memoria válido

    // Volvemos a liberar memoria una vez que ya no la necesitamos:
    free(pI);
    pI = NULL;

    return 0;
}
```

Es necesario comentar aquí que, aunque en los ejemplos anteriores no se haya utilizado la función `free()`, su uso se ha omitido porque la función aún no se había presentado, pero debemos utilizarla siempre para liberar la memoria reservada dinámicamente.

4. Arrays unidimensionales dinámicos

Como ya se ha expuesto, la principal característica de la memoria dinámica es que puede ser reservada en tiempo de ejecución. Esto la hace especialmente útil cuando necesitamos reservar una determinada cantidad de memoria (como un array o cadena de caracteres), pero no conocemos el número de elementos en tiempo de ejecución. Ante esta situación tenemos las siguientes opciones:

1. Crear un array con el número máximo de elementos que podemos llegar a necesitar. Esta solución es válida en todos los estándares de C, pero puede resultar en un desperdicio de memoria al no utilizar todas las posiciones reservadas.
2. Crear un array de longitud variable (VLA) con el tamaño requerido en tiempo de ejecución. Esta solución no desperdicia memoria al reservar el número exacto de elementos que necesitaremos, pero no todos los compiladores la soportan, ya que no está contemplada en todos los estándares de C, haciéndola menos compatible.
3. Crear un array en memoria dinámica. Esta solución permite ceñirnos exactamente al número de elementos necesarios, pudiendo incluso redimensionar el tamaño del array. Otra ventaja de esta solución es que la memoria dinámica se reserva en el montículo (o heap) en lugar de en la pila (o stack), por lo que su tamaño máximo no está limitado por el tamaño de la pila, sino por la memoria del sistema. Además,

este tipo de memoria es persistente, y puede ser utilizada hasta que decidamos liberarla (las dos primeras soluciones reservan memoria en la pila, que se libera al finalizar la función donde declaremos el array).

Para declarar y utilizar un array en memoria dinámica es suficiente con:

1. Declarar un puntero al tipo de datos del array.
2. Reservar un bloque de memoria con las funciones `malloc()` o `calloc()`.
3. Comprobar que la función ha retornado una dirección válida (en lugar de `NULL`).
4. Utilizar el array accediendo a sus elementos, bien con indexación de arrays, bien con aritmética de punteros.
5. Liberar la memoria utilizada con la función `free()`.

El siguiente ejemplo muestra el uso de un array dinámico. El código del ejemplo reserva memoria para un array utilizado para calcular la media de los números que inserte el usuario. Este array puede ser accedido utilizando aritmética de punteros (primer `for`) o utilizando indexación (segundo `for`). Después, cuando no se necesita acceder de nuevo a los elementos, se libera la memoria:

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

int main() {
   int num_elementos;
   int * p;
   float media = 0;
   printf("Inserta cuántos números quieres procesar:\n");
   scanf("%d", &num_elementos);

   // Reservamos el array en memoria dinámica:
   p = malloc(num_elementos * sizeof(int));
   // Verificamos que se ha podido reservar memoria:
   if(p == NULL) {
      printf("No se ha podido asignar memoria. Finalizando ejecución\n");
      return -1;
   }

   for(int i = 0; i < num_elementos; i++) {
      scanf("%d", (p+i)); // Recorremos el array dinámico con aritmética de punteros
   }

   for(int i = 0; i < num_elementos; i++) {
      media += p[i];  // Recorremos el array dinámico con indexación de arrays.
   }
   media /= num_elementos;

   // Liberamos la memoria reservada y apuntamos el puntero a NULL para evitar que apunte a una
dirección no válida:
   free(p);
   p = NULL;

   printf("La media es: %f\n", media);
   return 0;
}
```

Una de las ventajas del uso de memoria dinámica es que es el desarrollador el que elige cuando se reserva memoria y cuando se libera, a diferencia de la memoria estática (en la pila), que se reserva al iniciar la ejecución de una función y se libera al finalizar. Esto hace que podamos utilizar memoria dinámica para pasar punteros de una función a otra, como muestra el siguiente ejemplo:

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

/* Crea un array dinámico del número de elementos pasado como parámetros, y lo inicializa
 * leyendo valores de teclado */
int * crear_leer_array(int num_elementos)
{
   int * pArray = malloc(num_elementos * sizeof(int));
   if(pArray == NULL) {
      printf("No se ha podido asignar memoria. Finalizando ejecución\n");
      return NULL;
   }

   for(int i = 0; i < num_elementos; i++) {
      scanf("%d", &pArray[i]);
   }

   // Retorna el puntero del array dinámico
   return pArray;
}

int main() {
   int num_elementos = 3;
   int * p;

   // Almacenamos en p la dirección de memoria retornada por la función:
   p = crear_leer_array(num_elementos);
   // Verificamos que la función ha podido reservar memoria:
   if(p == NULL) {
      printf("No se ha podido asignar memoria. Finalizando ejecución\n");
      return -1;
   }

   // Mostramos el array
   for(int i = 0; i < num_elementos; i++) {
      printf("%d\t", p[i]);
   }

   // Liberamos la memoria reservada y apuntamos el puntero a NULL para evitar que apunte a una
dirección no válida:
   free(p);
   p = NULL;

   return 0;
}
```

5. Arrays bidimensionales dinámicos

También es posible reservar un array bidimensional (matriz) en memoria dinámica, permitiendo aprovechar las características de este tipo de memoria. Sin embargo, a diferencia de un array bidimensional declarado en la pila, en este caso sí se trata de un puntero a puntero propiamente dicho.

La idea en este caso es tener un puntero a puntero con la dirección de un array de punteros. Este array de punteros tendrá tantos elementos como filas tenga nuestra matriz, apuntando cada elemento del array de punteros a un array diferente (una fila) del tipo de datos de la matriz.

Para declarar un array bidimensional en memoria dinámica es necesario seguir los siguientes pasos:

1. Declarar un puntero a puntero al tipo de dato que queremos almacenar.
2. Reservar memoria dinámica para un array de punteros que tendrá tantos elementos como filas tenga nuestra matriz. Cada uno de estos punteros apuntará al primer elemento de cada una de las filas.
3. Reservar memoria dinámica para cada uno de los arrays que representarán las filas de la matriz. La dirección de memoria retornada se almacenará en el elemento correspondiente de punteros.

El siguiente ejemplo muestra la declaración de un array bidimensional con dimensiones `num_filas` y `num_columnas` para trabajar con elementos de tipo `int`:

```
int ** matriz;  // Declaración del puntero a puntero
int num_filas = 3;
int num_columnas = 2;

matriz = (int **) malloc(num_filas * sizeof(int *));  // Reservamos memoria para el array de punteros (este array tendrá num_filas elementos, un puntero para cada fila).

for (int i = 0; i < num_filas; i++)
{
   matriz[i] = (int *) malloc(num_columnas * sizeof(int));  // Reservamos la memoria correspondiente a cada una de las filas (cada una tendrá num_columnas elementos).
}
```

Una vez declarada la matriz en memoria dinámica podemos utilizarla como una matriz almacenada en la pila, accediendo a sus elementos utilizando indexación de arrays o aritmética de punteros. El siguiente ejemplo accede a los elementos de la matriz para inicializarla utilizando indexación de arrays, y después imprime dichos elementos utilizando aritmética de punteros:

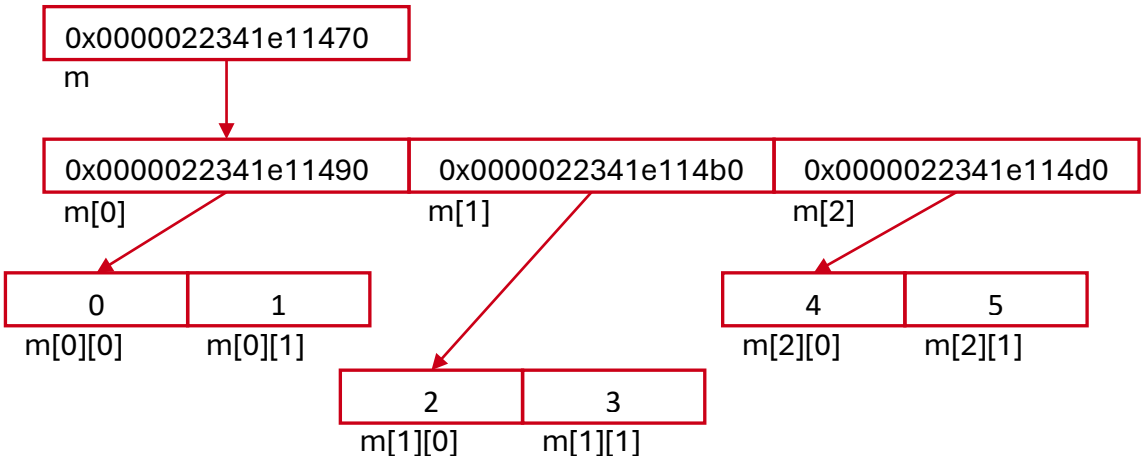
```
for(int i = 0; i < num_filas; i++)
{
   for(int j = 0; j < num_columnas; j++)
   {
      matriz[i][j] = i * num_filas + j;  // Accedemos al elemento i,j utilizando indexación
   }
}
```

```
for(int i = 0; i < num_filas; i++)
{
   for(int j = 0; j < num_columnas; j++)
   {
      printf("%d\t", *(*(matriz + i) + j));  // Accedemos al elemento i,j utilizando aritmética de punteros
   }
   printf("\n");
}
```

Finalmente, cuando no necesitemos utilizar más el espacio reservado, este debe ser liberado. En este caso, debemos liberar en primer lugar la memoria apuntada por cada uno de los punteros del array de punteros (utilizando un bucle `for`), liberando después la memoria apuntada por el puntero a punteros. El siguiente ejemplo muestra cómo liberar un array bidimensional en memoria dinámica:

```
for (int i=0;i<num_filas;i++)
{
   free(matriz[i]);  // Liberamos la memoria apuntada por cada uno de los punteros del array (filas)
}

free(matriz);  // Liberamos la memoria apuntada por el puntero a punteros (array con los punteros a cada una de las filas).
```

Es necesario resaltar aquí que este tipo de array bidimensional en memoria dinámica sí se trata de un puntero a puntero en el sentido estricto de la palabra. A diferencia de un array bidimensional en la memoria stack, en este caso sí hay un puntero a puntero en memoria que apunta a un array de punteros, casa uno de los cuales apuntando a una fila de la matriz. Por otro lado, aunque los elementos de cada una de las filas se almacenan de manera consecutiva en memoria, no sucede lo mismo con las diferentes filas, ya que la memoria se reserva de manera individual para cada una de ellas. La siguiente imagen muestra esta estructura para un array bidimensional `m` de 3 filas y 2 columnas como el creado en los ejemplos anteriores:



La siguiente tabla recoge las direcciones de memoria y el valor almacenado en dicha dirección en cada uno de los niveles de direccionamiento de la matriz m del ejemplo. Así mismo, también se muestra el tipo de valor almacenado y el identificador que nos permite acceder a cada valor:

Identificador	Tipo	Dirección en memoria	Valor
m	int **	0x00000057489ffba0	0x0000022341e11470
m[0]	int *	0x0000022341e11470	0x0000022341e11490
m[1]	int *	0x0000022341e11478	0x0000022341e114b0
m[2]	int *	0x0000022341e11480	0x0000022341e114d0
m[0][0]	int	0x0000022341e11490	0
m[0][1]	int	0x0000022341e11494	1
m[1][0]	int	0x0000022341e114b0	2
m[1][1]	int	0x0000022341e114b4	3
m[2][0]	int	0x0000022341e114d0	4
m[2][1]	int	0x0000022341e114d4	5

Al crear un array bidimensional en memoria dinámica se reserva memoria para cada una de las filas de manera individual, lo que nos da la opción de reservar un tamaño diferente para cada fila. Por ejemplo, el siguiente código reserva espacio para una matriz triangular inferior en memoria dinámica variando el número de elementos reservados en cada fila:

```
int ** matriz;  // Declaración del puntero a puntero
int num_filas = 3;
int num_columnas = 2;

matriz = (int **) malloc(num_filas * sizeof(int *));  // Reserva de memoria de los punteros a cada fila

for (int i=0; i<num_filas; i++)
{
   matriz[i] = (int *) malloc((i + 1) * sizeof(int));  // Reserva de memoria para los elementos de cada fila
}
```

En este caso, debemos tener cuidado de acceder a los elementos de manera correcta, ya que no todas las filas tendrán el mismo número de elementos. Por ejemplo, la primera fila (fila 0) de la matriz triangular creada en el anterior ejemplo tiene únicamente un elemento. Por otro lado, a la hora de liberar la memoria lo haremos de manera similar, ya que debemos liberar los punteros relativos a cada una de las filas y, finalmente, el puntero a puntero:

```
for (int i=0;i<num_filas;i++)
{
   free(matriz[i]);  // Liberamos la memoria apuntada por cada uno de los punteros del array (filas)
}

free(matriz);  // Liberamos la memoria apuntada por el puntero a punteros (array con los punteros a
cada una de las filas).
```

Tema 9: Estructuras y Tipos de Datos Enumerados

Las estructuras y los tipos de datos enumerados en C son fundamentales para organizar y manejar datos de manera eficiente y estructurada.

Las estructuras (`structs`) permiten agrupar diferentes tipos de datos bajo un mismo nombre, facilitando la representación de entidades complejas. Por ejemplo, se puede usar una estructura para representar a un estudiante, con campos como nombre, edad y matrícula. Esto permite manejar los datos relacionados de manera conjunta y acceder a cada atributo de forma individual. Las estructuras son especialmente útiles en proyectos donde se necesita manipular conjuntos de datos con propiedades variadas, y son la base para la programación orientada a objetos de otros lenguajes.

Por otro lado, los tipos de datos enumerados (`enums`) permiten definir un conjunto de valores constantes con nombres significativos, como los días de la semana o los meses del año. Estos valores son enteros simbólicos que ayudan a hacer el código más legible y menos propenso a errores, ya que restringen las variables a valores específicos. Al usar enumeraciones, el código es más intuitivo y se reduce el uso de números arbitrarios.

Tanto las estructuras como los tipos de datos enumerados son herramientas que facilitan la creación de código modular, claro y fácil de mantener, al mismo tiempo que optimizan el manejo de datos y mejoran la legibilidad del programa.

1. Estructuras

Una estructura es una colección de variables, denominadas miembros o campos, que pueden ser de diferentes tipos de datos. Estas variables se agrupan bajo un solo nombre, aunque es posible acceder a cada miembro de la estructura de forma individual. Las estructuras permiten trabajar con datos relacionados de una manera unificada, similar a cómo funcionan los registros en bases de datos.

Las estructuras permiten para representar y organizar datos complejos. Por ejemplo, en una aplicación que maneja información de estudiantes, podemos agrupar los atributos de un estudiante, como su nombre, edad y calificación, en una sola estructura en lugar de manejar variables individuales. Esto permite una organización de datos más lógica y fácil de gestionar.

En C, una estructura se define con la palabra clave `struct`. La definición establece los tipos de datos y los nombres de cada miembro que contiene la estructura. A diferencia de los tipos de datos simples, una estructura puede contener múltiples tipos de datos en su interior.

La sintaxis básica para definir una estructura en C es la siguiente:

```
struct nombre_estructura {
   tipo_dato1 nombre_miembro1;
   tipo_dato2 nombre_miembro2;
   // Otros miembros...
};
```

©2024 Diego Hortelano Haro, Gerardo Reyes Salgado, Manuel Rubio Sánchez. Algunos derechos reservados. Este documento se distribuye bajo la licencia "Atribución/Reconocimiento-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en https://creativecommons.org/licenses/by-sa/4.0/deed.es.

Donde:

- `struct`: indica que estamos definiendo una estructura.
- `nombre_estructura`: es el nombre de la estructura que estamos creando.
- Cada miembro de la estructura se define con un tipo de dato y un nombre de miembro.

1.1. Declaración de variables

Una vez que se ha definido la estructura, podremos declarar variables de ese tipo. La definición de la estructura es únicamente la definición de un nuevo tipo de dato. Si deseamos utilizar este nuevo tipo es necesario declarar variables de este (lo que reservará espacio en memoria para poder almacenar valores). La declaración puede hacerse en el momento de la definición o después de esta:

1.1.1. Declaración al definir la estructura

El siguiente ejemplo declara las variables `estudiante1` y `estudiante2`, del tipo `struct estudiante` en el momento de definir la estructura:

```
struct estudiante {
   char nombre[50];
   int edad;
   float calificacion;
} estudiante1, estudiante2;
```

1.1.2. Declaración después de definir la estructura

El siguiente ejemplo muestra la definición de la variable `alumno` del tipo `struct Estudiante`, la cual debe haber sido declarada previamente:

```
struct estudiante alumno;
```

1.3. Ejemplo completo

El siguiente ejemplo muestra la definición de una estructura, la declaración de variables de ese tipo y el acceso a sus miembros, aunque este último punto se detalla más adelante:

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

// Definición de la estructura:
struct estudiante {
   char nombre[50];
   int edad;
   float calificacion;
};

int main() {
   // Declaración de una variable de tipo struct Estudiante
   struct estudiante alumno;

   // Asignación de valores a los miembros de la estructura
   strcpy(alumno.nombre, "Carlos");
```

```
    alumno.edad = 20;
    alumno.calificacion = 8.5;

    // Impresión de los valores
    printf("Nombre: %s\n", alumno.nombre);
    printf("Edad: %d\n", alumno.edad);
    printf("Calificación: %.2f\n", alumno.calificacion);
    return 0;
}
```

En este ejemplo, `alumno` es una variable del tipo `struct estudiante` que tiene tres miembros: `nombre`, `edad` y `calificacion`. La asignación y acceso a los valores almacenados en los miembros de la estructura se realiza utilizando el operador punto (.).

2. Typedef

`typedef` es una palabra clave que permite crear alias o nombres alternativos para tipos de datos. Esta sentencia facilita el manejo de tipos complejos y largos, permitiendo nombrar un tipo de datos con una etiqueta más simple. Así, en lugar de repetir una definición compleja, podemos utilizar el alias definido para mejorar la legibilidad y simplicidad del código.

Por ejemplo, si estamos utilizando una estructura compleja, como una estructura de datos para representar a una persona, en vez de referirnos a ella con `struct persona`, podemos crear un alias y simplemente usar `st_persona`.

Entre los principales usos de `typedef` se encuentran:

- Hacer el código más legible: al usar alias para tipos complejos, es más fácil entender el propósito de ciertas variables.
- Reducir errores: evitar repetir definiciones complejas reduce las posibilidades de errores tipográficos y facilita cambios futuros.
- Simplificar la declaración de punteros: en lugar de escribir `int *`, podemos crear un tipo llamado `IntPointer` y usarlo en su lugar.

El siguiente ejemplo muestra el uso de la sentencia `typedef` combinado con una estructura, lo que simplifica el uso de la misma:

```
#include <stdio.h>
#include <string.h>
// Definimos una estructura y un alias para la misma
typedef struct persona {
    char nombre[50];
    int edad;
} st_persona;

int main() {
    st_persona p1;
    p1.edad = 25;
    strcpy(p1.nombre, "Juan");
    printf("Nombre: %s, Edad: %d\n", p1.nombre, p1.edad);
    return 0;
}
```

2.1. Omisión

Cuando usamos `typedef` al definir una estructura es posible omitir el nombre de la estructura. Esto simplifica el código, especialmente cuando no necesitamos utilizar el nombre original de la estructura más adelante, y solo requerimos un alias para facilitar su uso.

La ventaja de esta técnica es que permite declarar el tipo y crear un alias en una única declaración, lo que puede hacer el código más compacto y legible, especialmente en proyectos grandes o cuando la estructura se utiliza en múltiples lugares.

El siguiente ejemplo muestra la definición de la estructura omitiendo el nombre al utilizar `typedef`:

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

// Definición de la estructura st_alumno
typedef struct {
   char nombre[100];
   unsigned short edad;
   float peso;
   unsigned long matricula;
} st_alumno;

int main() {
   // Declaración e inicialización de una variable de tipo st_alumno:
   st_alumno alumno1;
   // Uso de la variable del tipo st_alumno
   // ...
   return 0;
}
```

Entre las principales ventajas de omitir el nombre de la estructura en su declaración encontramos:

1. Simplicidad: evita tener que repetir `struct` cada vez que usamos el tipo.
2. Legibilidad: al reducir el número de palabras clave, el código puede ser más claro, especialmente para los nuevos programadores.
3. Código limpio: al no tener que especificar el nombre de la estructura, se elimina información redundante.

Es importante resaltar que solamente es posible omitir el nombre de la estructura si el tipo solo se necesita a través de un alias, y no hay necesidad de hacer referencia directa a `struct nombre_estructura`. En caso de utilizar `typedef` para establecer un alias a una estructura, se considera buena práctica que dicho alias comience por "`s_`" o "`st_`" para indicar de manera rápida que se trata del alias de una estructura, como `st_estudiante` o `st_persona`.

Esta técnica es común en aplicaciones donde se define una estructura de datos para un propósito específico, como una lista o nodo, y queremos simplificar su uso.

3. Manejo de estructuras

Este punto profundiza en la gestión y manipulación de las estructuras en C, explorando conceptos clave para trabajar con sus miembros y utilizarlas en diversas aplicaciones.

3.1. Acceso a miembros

Para acceder a los miembros de una estructura, se utiliza el operador punto (`.`). Este operador permite acceder y modificar los campos de una estructura usando el nombre de la variable del tipo `struct` y el nombre del campo.

El siguiente ejemplo muestra la definición de una `struct` denominada `st_persona`, la declaración de una variable `persona1` de este tipo y el acceso a uno de sus miembros utilizando el operador punto (`.`):

```c
typedef struct {
   char nombre[50];
   int edad;
} st_persona;

int main() {
   st_persona persona1;
   persona1.edad = 30;  // Acceso a miembro edad
}
```

3.2. Operador flecha (->)

El operador flecha (`->`) se utiliza para acceder a los miembros de una estructura a través de un puntero. Este operador permite simplificar el acceso a los miembros sin necesidad de desreferenciar el puntero manualmente.

El siguiente ejemplo muestra el acceso a los miembros de una variable puntero del tipo `st_persona`. En este caso es posible utilizar directamente el operador `->`, lo que es totalmente equivalente a desreferenciar el puntero (utilizando el operador `*`) y acceder al miembro utilizando el operador punto (`.`):

```c
typedef struct {
   char nombre[50];
   int edad;
} st_persona;

int main() {
   st_persona persona1;
   st_persona * punt_persona1 = &persona1;

   // Acceso a miembro a través del puntero utilizando ->:
   punt_persona1->edad = 30;
   // Acceso a miembro desreferenciando el puntero:
   (*punt_persona1).edad = 30;
}
```

3.3. Copia de estructuras

En C, las estructuras se pueden copiar directamente usando el operador de asignación (=). Al copiar una estructura de esta forma, cada campo se copia individualmente. El siguiente ejemplo muestra el uso del operador de asignación para copiar dos estructuras:

```
st_persona persona1;
st_persona persona2;
persona2 = persona1;   // Copia todos los campos de persona1 a persona2
```

Sin embargo, debemos tener cuidado con esta situación, ya que cuando copiamos el contenido de una estructura a otra usando el operador de asignación (=), se realiza una copia superficial (shallow copy). Esto significa que cada campo de la estructura original se copia directamente en la nueva estructura. Por ello, si alguno de estos campos es un puntero, sólo se copia la dirección de memoria, no el contenido al que apunta.

Esto puede provocar problemas, ya que ambas estructuras terminarán apuntando al mismo bloque de memoria para el campo puntero. Cualquier cambio que se haga en el contenido de ese puntero en una estructura se reflejará en la otra, ya que comparten la misma dirección de memoria.

El siguiente ejemplo crea dos estructuras de tipo `st_persona`, copiando el contenido de una de ellas (`persona1`) en la otra (`persona2`) utilizando el operador de asignación, modificando su campo `nombre` y manteniendo el campo `edad`:

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

typedef struct {
   char * nombre; // Campo puntero
   int edad;
} st_persona;

int main() {
      st_persona persona1;  // Creamos e inicializamos la primera estructura
   persona1.nombre = (char *)malloc(50 * sizeof(char)); // Reservamos memoria para el nombre
   strcpy(persona1.nombre, "Juan");
   persona1.edad = 30;

   st_persona persona2 = persona1; // Copiamos persona1 en persona2 con el operador de asignación

   // Cambiar el nombre de persona2
   strcpy(persona2.nombre, "Ana");

   // Mostrar resultados
   printf("Nombre en persona1: %s\n", persona1.nombre);
   printf("Nombre en persona2: %s\n", persona2.nombre);

   // Liberar memoria asignada
   free(persona1.nombre);
   // No es seguro liberar persona2.nombre, ya que apunta al mismo lugar que persona1.nombre
   return 0;
}
```

Al ejecutar este código, obtendremos la siguiente salida:

```
Nombre en persona1: Ana
Nombre en persona2: Ana
```

A continuación, se detalla la explicación de por qué ocurre esto:

1. Cuando copiamos `persona1` en `persona2` con `st_persona persona2 = persona1;`, el campo `nombre` (que es un puntero) no crea una copia del contenido al que apunta, sino que simplemente copia la dirección de memoria.
2. Por lo tanto, tanto `persona1.nombre` como `persona2.nombre` apuntan a la misma dirección de memoria.
3. Al modificar `persona2.nombre` para que sea "`Ana`", también cambia el valor en `persona1.nombre`, ya que ambos apuntan a la misma dirección.
4. Finalmente, al liberar `persona1.nombre`, tendríamos un problema de doble liberación (double free) si intentamos liberar también `persona2.nombre`, ya que ambos apuntan a la misma dirección de memoria.

Para evitar este problema, necesitamos hacer una copia profunda (deep copy) del contenido al que apunta el puntero. Esto significa que debemos reservar memoria para el campo puntero de `persona2` y copiar el contenido de `persona1.nombre` en esa nueva área de memoria, tal como muestra el siguiente ejemplo:

```
st_persona persona2;
persona2.nombre = (char *)malloc(50 * sizeof(char)); // Reservar memoria para el nuevo nombre
strcpy(persona2.nombre, persona1.nombre); // Copiar el contenido
persona2.edad = persona1.edad; // Copiar el resto de los campos
```

En este caso, `persona1.nombre` y `persona2.nombre` apuntarán a bloques de memoria independientes, lo que evitará el problema de la copia superficial.

3.4. Estructuras en funciones

Al igual que el resto de variables, las estructuras pueden ser pasadas como parámetros a las funciones, ya sea por valor o por referencia. Al pasarlas por valor, se crea una copia ligera de la estructura, por lo que debemos tener cuidado si la estructura incluye campos que sean punteros, ya que seguirán accediendo a la dirección del campo de la estructura original; al pasarlas por referencia (usando punteros), la función puede modificar los datos originales.

El siguiente ejemplo muestra dos funciones que tienen un parámetro de entrada de tipo `st_persona`. La función `imprimir_persona` requiere un paso de parámetro por valor, mientras que la función `modificar_edad` utilizad un paso de parámetro por referencia:

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

typedef struct {
   char * nombre; // Campo puntero
   int edad;
} st_persona;
```

```c
void imprimir_persona(st_persona p)
{
   printf("Nombre: %s\n", p.nombre);
   printf("Edad: %d\n", p.edad);
}

void modificar_edad(st_persona * p, int nueva_edad)
{
   p->edad = nueva_edad;
}

int main() {
   // Creamos e inicializamos la variable de tipo st_persona
   st_persona persona1;
   persona1.nombre = (char *) malloc(50 * sizeof(char)); // Reservamos memoria para el nombre
   strcpy(persona1.nombre, "Juan");
   persona1.edad = 30;

   // Invocamos a las dos funciones definidas previamente:
   imprimir_persona(persona1); // La estructura se pasa como valor
   modificar_edad(&persona1, 35);  // La estructura se pasa como referencia

   free(persona1.nombre);
   return 0;
}
```

Usualmente, se considera buena práctica el paso por referencia de estructuras (en lugar del paso por valor), ya que el uso de memoria será menor. Así, para una estructura con múltiples campos, el paso por valor implica reservar de nuevo espacio para cada uno de los campos de la copia, mientras que un paso por referencia únicamente requiere la reserva del espacio necesario para una dirección de memoria.

3.4. Estructuras en arrays

Al igual que sucede con los tipos básicos, también es posible declarar arrays de un tipo definido por nosotros, como una estructura. Esto es útil cuando se necesita trabajar con un conjunto de datos estructurados del mismo tipo. Para ello, es suficiente con declarar una variable del tipo `struct` definido, e incluir entre corchetes (`[]`) el número de elementos del array. En este caso, cada uno de los elementos del array será una estructura completa que incluirá todos los campos definidos.

Al igual que sucede con cualquier otro array, podemos utilizar indexación de arrays, junto con el operador punto (`.`) para acceder a los miembros de cada una de las estructuras del array, o bien utilizar aritmética de punteros y el operador flecha (`->`). Otra opción posible es utilizar aritmética de punteros con el operador de desreferencia (`*`), en cuyo caso podremos acceder a los miembros de cada elemento del array con el operador punto (`.`).

El siguiente ejemplo muestra la declaración de un array de estructuras, y el acceso a los miembros de diferentes elementos de las tres formas descritas anteriormente, siendo todas ellas equivalentes:

```c
typedef struct {
   char * nombre;
   int edad;
} st_persona;

int main() {
   st_persona personas[3];

   personas[0].edad = 24;  // Acceso utilizando indexación de punteros y el operador "."
   (personas+1)->edad = 24; // Acceso utilizando aritmética de punteros y el operador "->"
   (*(personas + 2)).edad = 24; // Acceso utilizando aritmética de punteros y los operadores "*" y "."

   return 0;
}
```

Como puede apreciarse, el uso del operador punto (`.`) y del operador flecha (`->`) para acceder a los miembros de una estructura depende de si esta es una estructura directa, es decir, una instancia de la estructura, o de si se trata de un puntero a una estructura. En el primer caso, para acceder a los campos de una estructura directa, sea una variable o el elemento de un array (obtenido con indexación o con el uso de aritmética de punteros y el operador de desreferencia `*`) es necesario utilizar el operador punto (`.`). En cambio, si tenemos un puntero a una estructura, es decir, una dirección de memoria que apunta a la estructura, podemos acceder a los campos de la estructura apuntada directamente con el operador flecha (`->`).

4. Tipos de datos enumerados

Los tipos de datos enumerados o enumeraciones (`enum`), son un tipo de dato definido por el usuario en C que permite establecer un conjunto de constantes simbólicas (valores enteros) con nombres específicos. Estos tipos se utilizan para representar valores que se limitan a un rango definido de opciones, lo que ayuda a hacer el código más legible y fácil de mantener.

Por ejemplo, los días de la semana o los meses del año son conjuntos de valores definidos que se pueden representar mediante tipos enumerados. Al definir tipos `enum`, podemos evitar el uso de valores enteros "mágicos" en el código (como 0 para "Lunes", 1 para "Martes", etc.) y en su lugar utilizar nombres descriptivos. Las ventajas de utilizar este tipo de datos son:

- Mejora la legibilidad del código.
- Permite agrupar valores relacionados.
- Facilita la detección de errores, ya que el compilador puede advertir sobre el uso de valores fuera del rango definido en el `enum`.

4.1. Declaración

La declaración de un tipo enumerado en C se realiza con la palabra clave `enum`, seguida de un identificador que representa el nombre del tipo y un conjunto de valores entre llaves (`{}`). Cada uno de estos valores es una constante simbólica. La sintaxis básica de un tipo enumerado es la siguiente:

```
enum nombre_del_enum {
   valor1,
   valor2,
   valor3,
   // ... otros valores
};
```

Por defecto, los valores de un tipo enumerado comienzan desde 0 y se incrementan en 1 para cada elemento siguiente. El siguiente ejemplo muestra la definición de un tipo enumerado denominado `dias_semana`, que representa los días de la semana. En esta enumeración, la constante LUNES tiene el valor 0, MARTES el valor 1, y así sucesivamente:

```
enum días_semana {
   LUNES,   // 0
   MARTES,  // 1
   MIERCOLES, // 2
   JUEVES,  // 3
   VIERNES, // 4
   SABADO,  // 5
   DOMINGO  // 6
};
```

Sin embargo, también se pueden asignar valores específicos a algunas o a todas las constantes de la enumeración. Esto es útil cuando queremos que un elemento del `enum` tenga un valor específico diferente del predeterminado. En el siguiente ejemplo se define el tipo `enum meses_año`, en que se le asigna a la constante `ENERO` el valor 1, por lo que el resto de los meses se incrementa en 1 automáticamente. Así, `FEBRERO` tiene el valor 2, `MARZO` el valor 3, y así sucesivamente hasta `DICIEMBRE` que tiene el valor 12:

```
enum meses_año {
   ENERO = 1,
   FEBRERO,
   MARZO,
   ABRIL,
   MAYO,
   JUNIO,
   JULIO,
   AGOSTO,
   SEPTIEMBRE,
   OCTUBRE,
   NOVIEMBRE,
   DICIEMBRE
};
```

4.2. Uso en el código

Una vez declarado un tipo enumerado, se puede usar para declarar variables que solo pueden contener uno de los valores definidos en el `enum`. Esto facilita el control y la validación de datos.

En el siguiente ejemplo, la variable `hoy` puede almacenar únicamente los valores definidos en `dias_semana`. Esto hace que el código sea más seguro y legible, ya que se eliminan los

valores arbitrarios. Además, es posible comparar los valores directamente con las constantes definidas en la propia enumeración:

```
enum dias_semana {
   LUNES,
   MARTES,
   MIERCOLES,
   JUEVES,
   VIERNES,
   SABADO,
   DOMINGO
};

int main() {
   enum dias_semana hoy = MIERCOLES;
   if (hoy == MIERCOLES) {
      printf("Hoy es miércoles.\n");
   }
   return 0;
}
```

Al igual que con `struct`, `typedef` también puede utilizarse para definir alias de los tipos enumerados. En este caso, es común que dichos nombres comiencen por "`e_`" para indicar que se trata de un tipo enumerado. El siguiente ejemplo muestra la definición de un tipo enumerado y la declaración de una variable de este tipo utilizando `typedef`:

```
typedef enum dias_semana {
   LUNES,
   MARTES,
   MIERCOLES,
   JUEVES,
   VIERNES
} e_dias;

int main() {
   e_dias hoy = MIERCOLES;

   return 0;
}
```

Y, de igual manera, podemos omitir el nombre de la enumeración, dando directamente su alias, tal y como muestra el siguiente ejemplo:

```
typedef enum {
   LUNES,
   MARTES,
   MIERCOLES,
   JUEVES,
   VIERNES
} e_dias;
```

Tema 10: Ficheros

Hasta ahora hemos utilizado la entrada y salida estándar (entrada mediante el teclado y salida mediante pantalla), la cual se realiza mediante los buffers de los flujos de entrada y salida (`stdin` y `stdout`, respectivamente). Aunque este tipo de entrada/salida permite que el usuario se comunique con nuestros programas, tiene limitaciones importantes.

Por una parte, ciertas entradas están compuestas por grandes cantidades de datos, pudiendo provocar errores por parte del usuario al intentar escribirlas manualmente. También es posible que la entrada de nuestro programa se encuentre ya en un fichero, el cual puede haber sido creado manualmente o como salida de otro programa. Por otra parte, al utilizar la salida estándar los datos no son persistentes, no pudiendo almacenarlos al finalizar el programa. Esto puede ser un problema en determinadas ocasiones, en las que necesitemos almacenar los resultados obtenidos para procesarlos o simplemente consultarlos más adelante.

Los ficheros permiten almacenar y leer datos de un almacenamiento permanente (como el disco duro), eliminando algunas restricciones de las mencionadas anteriormente para la entrada/salida estándar. En este tema se introduce el uso de ficheros en C.

1. Definición

Un fichero o archivo es una colección de datos almacenada en un dispositivo de almacenamiento persistente, como el disco duro (memoria secundaria), y que permite conservar información a largo plazo, aunque finalice nuestro programa. Esto permite que los datos puedan ser leídos o escritos en varias ejecuciones del programa, o incluso que lo hagan diferentes programas (por ejemplo, la salida de un programa puede ser la fuente de entrada de otro programa).

Los ficheros se identifican utilizando su nombre y su ruta de acceso completa, que incluye el nombre de todos los directorios en los que se encuentra. Dentro de un mismo directorio el nombre de un archivo debe ser único, lo que significa que cada fichero se identifica de manera inequívoca, permitiendo un acceso seguro y preciso a los datos deseados.

En C, un fichero es un flujo de bytes organizados, que puede ser texto o binario:

- Ficheros de texto: contiene datos en formato legible para los humanos como texto plano (caracteres ASCII), interpretándose como una secuencia de líneas de caracteres. Esto permite que los ficheros de este tipo sean leídos y escritos tanto por personas como por programas. El acceso a los ficheros de texto se realiza de manera secuencial, lo que implica que sea necesario leer todas las posiciones previas a una determinada posición para leer dicha posición.
- Ficheros binarios: contiene datos en formato binario, lo que resulta más eficiente en términos de espacio en memoria y velocidad de procesamiento. Sin embargo, estos ficheros no son directamente legibles para las personas. El acceso a los ficheros binarios se realiza de manera directa, pudiendo posicionarnos en cualquier dato del fichero sin pasar por las posiciones que lo preceden.

©2024 Diego Hortelano Haro, Gerardo Reyes Salgado, Manuel Rubio Sánchez. Algunos derechos reservados. Este documento se distribuye bajo la licencia "Atribución/Reconocimiento-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en https://creativecommons.org/licenses/by-sa/4.0/deed.es.

La siguiente tabla recoge las principales diferencias entre los ficheros de texto y los ficheros binarios:

Característica	Ficheros de texto	Ficheros binarios
Almacenamiento	Texto plano (ASCII)	Formato binario
Legibilidad	Legibles por personas	No legibles por personas
Tamaño de fichero	Mayor tamaño debido a caracteres adicionales	Menor espacio
Velocidad de procesamiento	Lenta, ya que requiere conversión	Más rápida, ya que almacenan en un formato procesable directamente.
Funciones de lectura/escritura	`fprintf`, `fscanf`, `fgets`, `fputs`	`fwrite`, `fread`
Portabilidad	Mayor portabilidad (conversión automática de caracteres entre sistemas)	Menor portabilidad, ya que dependen del sistema y del tamaño de los datos

En los siguientes puntos se muestran funciones para el manejo de ficheros de texto y ficheros binarios en C, todas ellas disponibles en la biblioteca de entrada/salida estándar (`stdio.h`).

2. Apertura de ficheros

La apertura de un fichero en C establece el canal de comunicación entre el programa y el fichero en cuestión. Para ello, se utiliza la función `fopen`, cuya sintaxis es:

```
FILE * p_fichero = fopen(const char * nombre_fichero, const char * modo)
```

Donde:

- `p_fichero`: puntero al descriptor del fichero, de tipo `FILE`. Si no se puede abrir el fichero, la función retornará `NULL`, permitiéndonos comprobar si ha habido algún error. El tipo FILE es una estructura interna (no podemos acceder directamente a sus campos) que contiene la información necesaria para gestionar las operaciones de entrada/salida en ficheros. Aunque los campos de esta estructura no están especificados por el estándar y pueden variar en función del compilador y sistema operativo, usualmente se encuentran los siguientes punteros:
  - Puntero al buffer intermedio, utilizado para almacenar temporalmente los datos leídos/escritos en el fichero.
  - Posición actual en el fichero (cursor), que indica dónde se realizará la siguiente lectura/escritura.
  - Estado de la operación, que aporta información sobre si la operación se ejecutó correctamente o si ocurrió algún error. Destaca el indicador de error EOF (End Of File), que permite saber si se ha leído el fichero completo.
  - Información del tipo de fichero y del modo de apertura utilizado.
- `nombre_fichero`: cadena de caracteres con el nombre del fichero que queremos abrir. Podemos hacer referencia con la ruta relativa (desde donde se encuentra nuestro proyecto) o utilizando la ruta absoluta (desde el directorio raíz del sistema).
- `modo`: cadena de caracteres que especifica el tipo de fichero (texto o binario) y los permisos de apertura.

Los diferentes modos de apertura de un fichero se muestran en la siguiente tabla:

Fichero de texto	Fichero binario	Descripción
`"r"`	`"rb"`	Abre el archivo para lectura. El archivo debe existir, si no, la función retornará `NULL`.
`"r+"`	`"r+b"`	Abre el archivo para lectura y escritura. El archivo debe existir, si no, la función retornará `NULL`.
`"w"`	`"wb"`	Abre el archivo para escritura. Si existe, se sobrescribe, si no, se crea uno nuevo.
`"w+"`	`"w+b"`	Abre el archivo para lectura y escritura. Si existe, se sobrescribe, si no, se crea uno nuevo.
`"a"`	`"ab"`	Abre un archivo para escritura en modo anexado. Si existe, los datos se añaden al final, si no, se crea uno nuevo.
`"a+"`	`"a+b"`	Abre un archivo para lectura y escritura en modo anexado. Si existe, los datos se añaden al final, si no, se crea uno nuevo.
`"x"`		Abre un archivo de texto para escritura exclusiva. El archivo no debe existir (creándose con la llamada a la función). Si ya existe, la función falla y retorna `NULL`.

A la vista de los modos de apertura, podemos encontrar los siguientes errores al intentar abrir un fichero:

- Intentamos abrir un fichero en modo lectura o lectura y escritura (`"r"`, `"rb"`, `"r+"` o `"r+b"`) y el fichero no existe.
- Intentamos abrir un fichero en modo escritura (`"w"`, `"wb"`, `"a"`, `"ab"`, `"a+"`, `"a+b"` o `"x"`) y no tenemos permisos de escritura sobre el fichero.
- Intentamos abrir un fichero que no existe en modo escritura (`"w"`, `"wb"`, `"a"`, `"ab"`, `"a+"`, `"a+b"` o `"x"`) y no tenemos permisos de escritura en el directorio donde se va a crear.
- Intentamos abrir un fichero de texto para escritura exclusiva (`"x"`) y el archivo ya existe.

El siguiente ejemplo muestra la apertura de un fichero de texto y un fichero binario. El fichero de texto se especifica utilizando una ruta relativa, por lo que se tendrá en cuenta la ruta desde el directorio del ejecutable de nuestro programa. Este fichero se abre en modo lectura y escritura con `"r+"`, por lo que el fichero debe existir previamente. Por otra parte, el fichero binario se especifica utilizando una ruta absoluta desde el directorio raíz del sistema (destaca aquí el uso de doble barra `"\\"`, al tratarse de un carácter de escape). Este segundo fichero se abre en modo de escritura con `"wb"`, por lo que el fichero se sobrescribirá si ya existe:

```
#include <stdio.h>
int main() {
   FILE * fichero_bin;
   // Apertura de un fichero de texto utilizando una ruta relativa en modo lectura y escritura.
   FILE * fichero_texto = fopen("texto.txt", "r+");
   if (fichero_texto == NULL) {
      printf("No se pudo abrir el archivo de texto.\n");
      return -1;
   }
```

```
    // Apertura de un fichero binario utilizando una ruta absoluta en modo escritura.
    fichero_bin = fopen("C:\\Users\\diego.hortelano\\CLionProjects\\ejemplo\\binario.bin", "wb");
    if (fichero_bin == NULL) {
        printf("No se pudo abrir el archivo de texto.\n");
        return -1;
    }
    return 0;
}
```

3. Cierre de ficheros

Una vez que hayamos terminado de utilizar un fichero, debemos cerrarlo para liberar recursos (asociados al fichero), proteger los datos (asegurándose que todos los datos en el buffer intermedio se escriben en el fichero antes de finalizar nuestro programa) y evitar el límite de ficheros abiertos por proceso existentes en muchos sistemas. Para ello podemos utilizar la función `fclose()`, cuya sintaxis es:

```
int fclose(FILE * p_fichero)
```

Donde:

- `p_fichero`: es el puntero a la estructura `FILE` del fichero que queremos cerrar.

Esta función retorna un valor entero, que indicará si el fichero se cerró correctamente (si el valor es 0) o si ha habido algún error al cerrar el fichero (con un valor diferente de 0, dependiendo del error ocurrido).

El siguiente ejemplo muestra dos llamadas a la función `fclose()` para cerrar los ficheros abiertos en el ejemplo anterior con `fopen()`:

```
if(fclose(fichero_texto) != 0)
{
    printf("Error al cerrar el fichero de texto");
}

if(fclose(fichero_bin) != 0)
{
    printf("Error al cerrar el fichero binario");
}
```

4. Final de fichero

En ciertas ocasiones es posible conocer el contenido de un fichero (o al menos la cantidad de datos almacenados en dicho fichero). Sin embargo, en otras ocasiones puede ser necesario leer todo el fichero, sin conocer de antemano la cantidad de datos que puede contener. En este contexto destaca la función `feof()`, la cual permite verificar si se ha llegado al final de un fichero (`EOF`). La sintaxis de esta función es la siguiente:

```
int feof(FILE * fichero)
```

Donde:

- `fichero`: es el puntero a la estructura de tipo FILE del que queremos verificar si hemos llegado al final.

Esta función retorna un entero, un valor distinto de 0 (verdadero en C) si hemos alcanzado el final del fichero (`EOF`) y 0 (falso) en caso contrario, indicando que podemos continuar leyendo.

Esta función puede utilizarse tanto con ficheros de texto como con ficheros binarios. A continuación, se muestra un ejemplo de uso de esta función con los archivos anteriores (por supuesto, la función `feof()` debe utilizarse antes de cerrar el fichero con la función `fclose()`):

```
if(feof(fichero_texto)) {
   printf("Se alcanzó el final del fichero de texto.\n");
}

if (feof(fichero_bin)) {
   printf("Se alcanzó el final del fichero de texto.\n");
}
```

5. Lectura y escritura en ficheros de texto

Para leer y escribir en ficheros de texto podemos utilizar las funciones `fscanf()` y `fprintf()`, similares a las funciones `scanf()` y `printf()` que ya conocemos, con la salvedad de que debemos indicar el fichero del cual queremos leer o escribir.

Comenzando por la **lectura**, la función `fscanf()` tiene la siguiente sintaxis:

```
int fscanf(FILE * fichero, const char * formato, …)
```

Donde:

- `fichero`: puntero a la estructura de tipo `FILE` del fichero del que queremos leer. Este fichero debe estar abierto en un modo que permita la lectura (como `"r"`, `"r+"`, `"w"` o `"w+"`).
- `formato`: cadena de caracteres que indica el formato de los datos que se leerán. Incluye códigos de control y caracteres de escape (como `%d` para enteros o `%f` para float), exactamente igual que la función `scanf()`.
- …: lista de variables (punteros) donde se almacenarán los datos leídos.

La función `fscanf()` retorna un entero con el número de elementos correctamente leídos y almacenados en las variables, o la constante `EOF` si se llega al final del fichero u ocurre un error de lectura al intentar leer algún dato.

A continuación, se muestra un ejemplo de un programa que utiliza la función `fscanf()` para leer caracteres de tres en tres, hasta leer todos los datos del fichero:

```
#include <stdio.h>

int main() {
   FILE * file = fopen("fichero_texto.txt", "r");
   if (file == NULL) {
      printf ("Error al abrir el fichero\n ");
      return 1;
   }
```

```
    char nombre[20];
    int edad;
    float altura;
    while (fscanf(file, "%s %d %f", nombre, &edad, &altura) == 3) {
        printf("Nombre: %s, Edad: %d, Altura: %.2f\n", nombre, edad, altura);
    }

    fclose(file);
    return 0;
}
```

También es posible utilizar la constante `EOF` en combinación con `fscanf()`, aunque es más seguro comprobar el número de elementos leídos en cada iteración, como en el ejemplo anterior. A continuación, se muestra un ejemplo similar al anterior, pero utilizando la constante `EOF` en la condición:

```
#include <stdio.h>

int main() {
    FILE *file = fopen("fichero_texto.txt", "r");
    if (file == NULL) {
        printf("Error al abrir el fichero\n");
        return 1;
    }

    char nombre[20];
    int edad;
    float altura;
    while (fscanf(file, "%s %d %f", nombre, &edad, &altura) != EOF) {
        printf("Nombre: %s, Edad: %d, Altura: %.2f\n", nombre, edad, altura);
    }

    fclose(file);
    return 0;
}
```

En cuanto a la **escritura** de datos en un fichero de texto, podemos hacer uso de la función `fprintf()` para escribir datos con formato de manera similar a como lo hace la función `printf()` en la salida estándar. La sintaxis de la función `fprintf()` es:

```
int fprintf(FILE * fichero, const char * formato, …)
```

Donde:

- `fichero`: puntero a la estructura de tipo `FILE` del fichero donde queremos escribir. Este fichero debe estar abierto en un modo que permita la escritura (como `"w"`, `"a"`, `"r+"`, "w+" o `"a+"`).
- `formato`: cadena de caracteres que indica el formato de los datos a escribir. Puede incluir códigos de control y caracteres de escape, además de otros caracteres, de manera similar a la función `printf()`.
- …: lista de parámetros que se escribirán en el fichero siguiendo el formato de la cadena anterior.

La función `fprintf()` retorna un entero indicando el número de caracteres escritos con éxito en el fichero un valor negativo si ocurre algún error de escritura.

A continuación, se muestra un ejemplo de uso de la función `fprintf()`, comprobando en cada caso que los datos se han podido escribir correctamente. Si se detecta que no ha sido así (la función retornará un valor negativo), cerramos el fichero y finalizamos el programa:

```c
#include <stdio.h>

int main() {
    FILE * file = fopen("fichero_texto.txt", "w");
    if (file == NULL) {
        printf("Error al abrir el fichero\n ");
        return 1;
    }

    const char * nombres[3] = {"Ana", "Luis", "Carlos"};
    int edades[3] = {20, 22, 19};
    float notas[3] = {8.5, 9.0, 7.8};

    for (int i = 0; i < 3; i++) {
        // Comprobamos que fprintf() no retorne un valor negativo
        if (fprintf(file, "Nombre: %s, Edad: %d, Nota: %.1f\n", nombres[i], edades[i], notas[i]) < 0) {
            printf("Error al escribir en el fichero\n");
            fclose(file);
            return 1;  // Salimos del programa si ocurre un error
        }
    }

    fclose(file);
    return 0;
}
```

También es posible utilizar las funciones `fgets()` y `fputs()` ya conocidas para leer y escribir de un fichero, aunque en este caso tendremos que utilizar cadenas de caracteres, sin poder especificar el formato de los datos. Además, tendremos que hacer referencia al puntero del fichero en lugar de al buffer de entrada/salida estándar. A continuación, se muestra un ejemplo del uso de estas funciones con ficheros:

```c
#include <stdio.h>

int main() {
    // Abrimos el archivo en modo "w+" para escribir y leer
    FILE * file = fopen("fichero_texto.txt", "w+");
    if (file == NULL) {
        printf("Error al abrir el fichero\n");
        return 1;
    }

    // Escribir líneas de texto en el archivo usando fputs()
    fputs("Primera línea de texto.\n", file);
    fputs("Segunda línea de texto.\n", file);
    fputs("Tercera línea de texto.\n", file);
```

```
    // Movemos el puntero al inicio del archivo para poder leer desde el principio
    rewind(file);

    // Leer y mostrar cada línea usando fgets()
    char buffer[100];
    printf("Contenido del archivo:\n");
    while (fgets(buffer, 100, file) != NULL) {
        printf("%s", buffer);  // Imprime cada línea leída
    }

    // Cerrar el archivo después de leer y escribir
    fclose(file);

    return 0;
}
```

Destaca aquí el uso de la función `rewind()`, la cual vuelve a colocar el cursor al inicio del fichero para poder leer los datos que acabamos de escribir en él. Esta función requiere como parámetro el puntero a la estructura que identifica al fichero en cuestión.

6. Lectura y escritura en ficheros binarios

Si abrimos el fichero en modo binario podremos leer y escribir directamente los bytes de las variables, aumentando la eficiencia de nuestro programa. Para ello podemos hacer uso de las funciones `fread()` y `fwrite()`.

La **lectura** de un fichero binario puede realizarse utilizando la función `fread()`, la cual permite leer bloques de datos binarios desde un fichero y almacenarlos en memoria. Esta función resulta especialmente útil para leer datos en bloques, como estructuras o arrays. Su sintaxis es:

```
size_t fread(void * p_mem, size_t tamaño, size_t contador, FILE * fichero)
```

Donde:

- `p_mem`: puntero a la memoria donde se almacenarán los datos leídos.
- `tamaño`: tamaño de cada elemento a leer (en bytes). Para obtener este tamaño puede utilizarse el operador `sizeof()`.
- `contador`: número de elementos a leer del tamaño indicado.
- `fichero`: puntero a la estructura de tipo `FILE` relativa al fichero del que queremos leer. Este fichero debe estar abierto en un modo que permita la lectura.

Esta función retorna el número de elementos leídos correctamente. Este número puede tener el mismo valor que `contador` si se han leído todos los elementos correctamente, o un número menor si se alcanza el final de fichero o si ocurre un error de lectura.

A continuación, se muestra un ejemplo del uso de `fread()` para leer un fichero binario que contiene un array de números enteros, almacenándolos en un array. Este array se ha declarado en memoria estática, pero podría haberse declarado en memoria dinámica:

```c
#include <stdio.h>

int main() {
   FILE * file = fopen("fichero_bin.bin", "rb");
   if (file == NULL) {
      printf("Error al abrir el fichero\n");
      return 1;
   }

   int buffer[5];  // Array donde se almacenarán los datos leídos

   // Leemos 5 elementos de tipo int desde el fichero
   int elementos_leidos = fread(buffer, sizeof(int), 5, file);
   if (elementos_leidos != 5) {
      if (feof(file)) {  // Comprobamos si se ha llegado al final del fichero
         printf("Se alcanzó el final del fichero.\n");
      } else if (ferror(file)) {  // Comprobamos si ha habido algún error
         printf("Error al leer del fichero.\n");
      }
   }

   // Mostrar los datos leídos
   for (size_t i = 0; i < elementos_leidos; i++) {
      printf("Número %d: %d\n", i + 1, buffer[i]);
   }

   fclose(file);
   return 0;
}
```

A la hora de **escribir** bloques de datos binarios en un fichero podemos utilizar `fwrite()`. Esta función permite trabajar con ficheros binarios de manera sencilla, así como escribir grandes bloques de datos en una sola operación. La sintaxis de `fwrite()` es:

```
size_t fwrite(const void * p_mem, size_t tamaño, size_t cont, FILE * fichero)
```

Donde:

- `p_mem`: puntero a la memoria donde se encuentran almacenados los datos que se van a escribir en el fichero.
- `tamaño`: tamaño de cada elemento a leer (en bytes). Para obtener este tamaño puede utilizarse el operador `sizeof()`.
- `cont`: número de elementos a leer del tamaño indicado.
- `fichero`: puntero a la estructura de tipo `FILE` relativa al fichero en el que queremos escribir los datos. Este fichero debe estar abierto en un modo que permita la escritura de datos.

La función `fwrite()` retorna el número de elementos que se escribieron correctamente. Si este número tiene un valor menor que `cont`, quiere decir que ha ocurrido un error de escritura al escribir un elemento o que no había espacio suficiente para terminar la escritura.

A continuación, se muestra un ejemplo del uso de la función `fwrite()`, la cual se utiliza para escribir los cinco elementos de un array definido en el propio código:

```c
#include <stdio.h>

int main() {
   FILE * file = fopen("fichero_bin.bin", "wb");
   if (file == NULL) {
      printf("Error al abrir el fichero\n");
      return 1;
   }

   int numeros[] = {10, 20, 30, 40, 50};
   size_t elementos_escritos = fwrite(numeros, sizeof(int), 5, file);

   if (elementos_escritos != 5) {
      printf("Error al escribir en el fichero\n");
   } else {
      printf("Se escribieron %u elementos correctamente.\n", elementos_escritos);
   }

   fclose(file);
   return 0;
}
```

7. Desplazamiento del cursor dentro del fichero

En ocasiones, podemos necesitar desplazar el cursor de un fichero a una posición determinada, bien para acceder directamente a una posición determinada para leer o escribir los datos desde esa posición, bien para volver a situarnos en otra posición después de realizar una operación. Por ejemplo, podríamos escribir unos datos determinados, mover el cursor a la posición inicial de la escritura y leer los datos escritos para verificarlos.

En uno de los ejemplos anteriores se ha incluido la función `rewind()`, que permite situar el cursor en la primera posición del fichero. También es posible mover el cursor a cualquier ubicación de un fichero utilizando la función `fseek()`. Su sintaxis es:

```
int fseek(File * fichero, long desplazamiento, int origen)
```

Donde:

- `fichero`: puntero a la estructura de tipo `FILE` relativa al fichero en el que queremos desplazar el cursor.
- `desplazamiento`: número de bytes a desplazarnos desde la posición especificada por el parámetro `origen`. Este número puede ser tanto positivo como negativo.
- `origen`: punto de referencia para el desplazamiento. Admite tres valores:
  - `SEEK_SET`: inicio del fichero.
  - `SEEK_CUR`: posición actual del cursor en el fichero.
  - `SEEK_END`: final del fichero.

La función `fseek()` retorna un entero con valor 0 si el movimiento del cursor se ha realizado correctamente, y con valor -1 en caso de error (por ejemplo, si la posición donde intentamos movernos se encuentra fuera de los límites del fichero.

El siguiente ejemplo utiliza la función `fseek()` para desplazar el cursor dos elementos enteros, permitiendo leer directamente el número entero almacenado en la tercera posición:

```c
#include <stdio.h>

int main() {
    FILE * file = fopen("fichero_bin.bin", "wb");
    if (file == NULL) {
        printf("Error al abrir el fichero\n");
        return 1;
    }

    int dato;

    // Mover el puntero de posición 2 elementos (2 * sizeof(int) bytes) desde el inicio
    if (fseek(file, 2 * sizeof(int), SEEK_SET) != 0) {
        printf("Error al mover el puntero en el fichero\n");
        fclose(file);
        return 1;
    }

    // Leer el dato en la posición deseada
    fread(&dato, sizeof(int), 1, file);
    printf("Dato en la tercera posición: %d\n", dato);

    fclose(file);
    return 0;
}
```

El siguiente ejemplo hace uso de la función `fseek()` para desplazar el cursor al final del fichero. A continuación, se utiliza la función `ftell()` para obtener la posición del cursor. Al tratarse de un fichero en binario, la posición final será el tamaño del fichero en bytes:

```c
#include <stdio.h>

int main() {
    FILE * file = fopen("fichero_bin.bin", "wb");
    if (file == NULL) {
        printf("Error al abrir el fichero\n");
        return 1;
    }

    // Mover el puntero al final del fichero
    fseek(file, 0, SEEK_END);

    // Obtener la posición actual del puntero, que indica el tamaño del fichero
    long tam = ftell(file);
    printf("Tamaño del archivo: %ld bytes\n", tam);

    fclose(file);
    return 0;
}
```

La función `fseek()` puede utilizarse tanto con ficheros de texto como con ficheros binarios. Sin embargo, no se recomienda el uso con ficheros de texto, ya que su comportamiento puede ser impredecible al estar los datos almacenados de diferentes maneras en función del sistema operativo.

Anexo I: Instalación y configuración de CLion en Windows

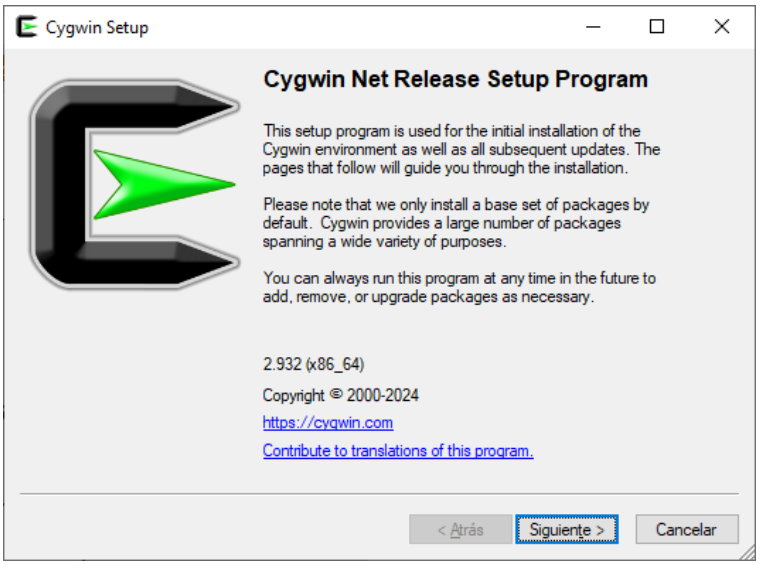
CLion requiere la instalación de una serie de herramientas y configuraciones para compilar, depurar y gestionar proyectos de C de manera correcta en Windows. Entre estas herramientas, necesitamos un compilador y un depurador. Para esto podemos utilizar herramientas software diferentes, como Cygwin o MinGW.

1. Instalación de Cygwin

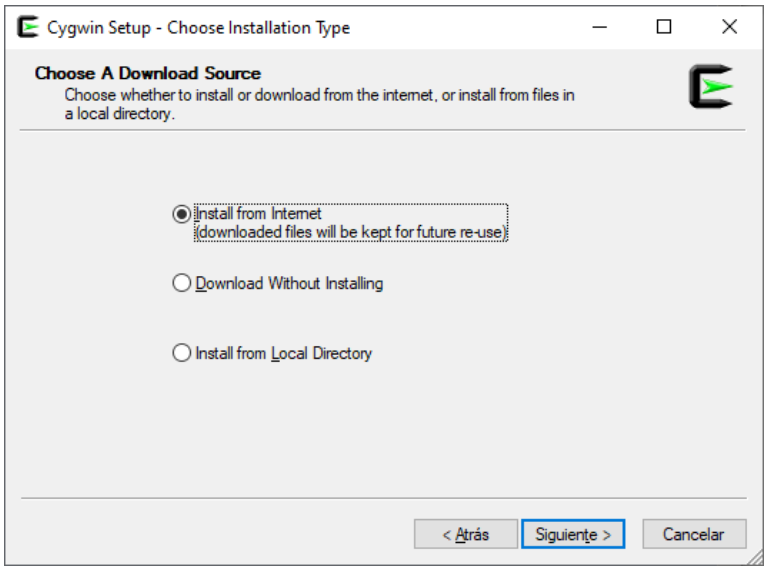
Tomando como ejemplo Cygwin, podemos descargarlo desde su página (https://www.cygwin.com/, seleccionando la opción "*Install Cygwin*" y "*setup-x86-64.exe*").

Una vez descargado, debemos ejecutar el archivo descargado, apareciendo la siguiente interfaz:

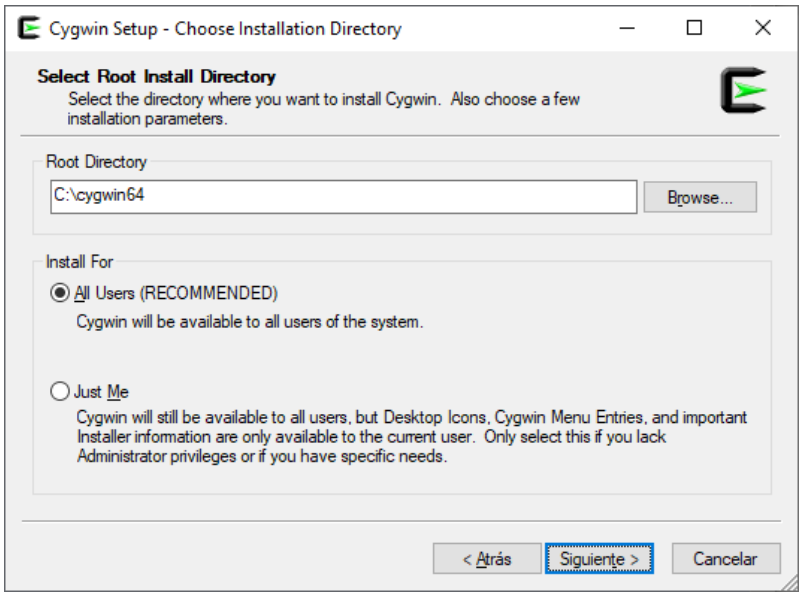
©2024 Diego Hortelano Haro, Gerardo Reyes Salgado, Manuel Rubio Sánchez. Algunos derechos reservados. Este documento se distribuye bajo la licencia "Atribución/Reconocimiento-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en https://creativecommons.org/licenses/by-sa/4.0/deed.es.



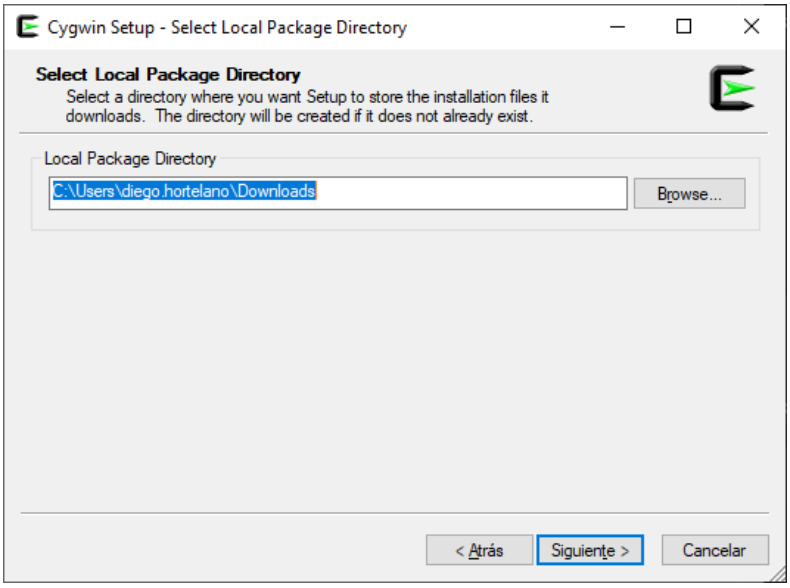
Seleccionamos "Siguiente", apareciendo la siguiente pantalla:



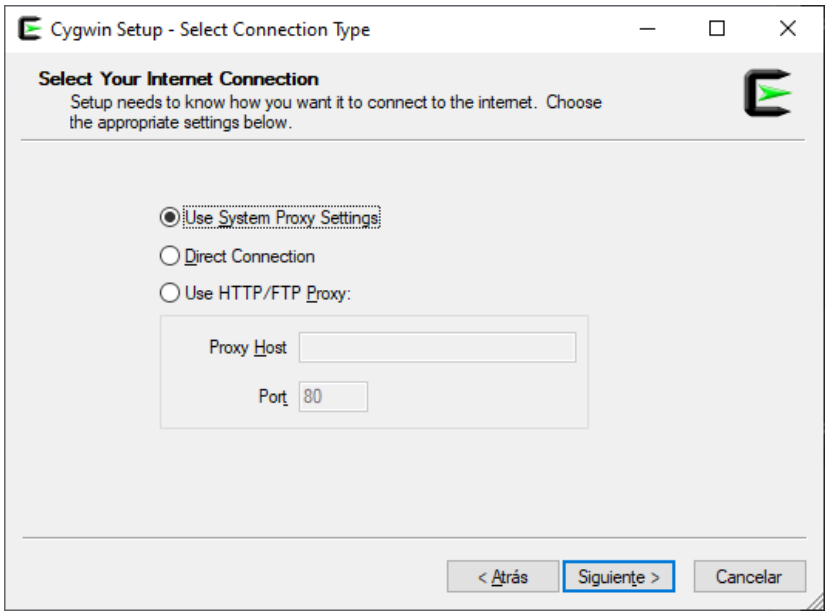
Aquí marcamos la opción "Install from Internet" si no está marcada y, de nuevo, "Siguiente", mostrándonos la siguiente pantalla:



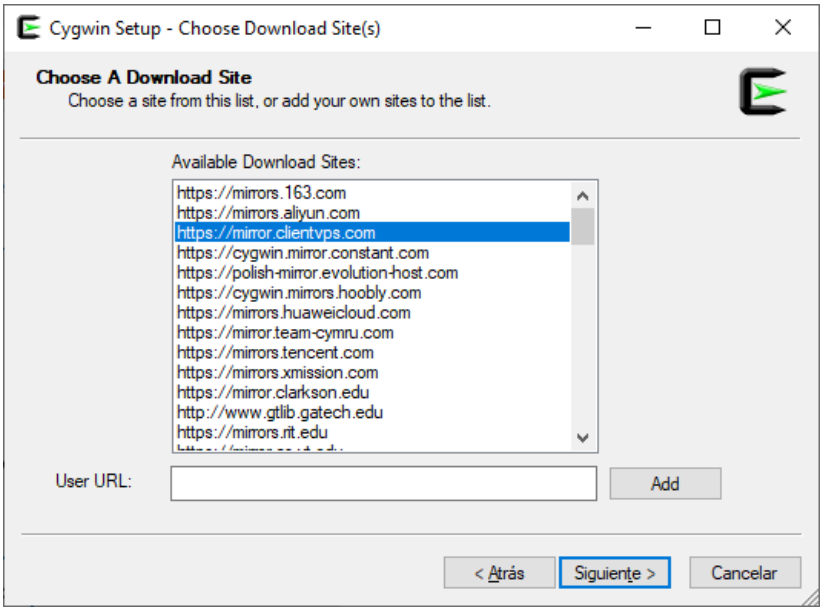
Se recomienda dejar el directorio de instalación por defecto para que CLion lo reconozca automáticamente, así como la opción de instalación para todos los usuarios. Si pulsamos "*Siguiente*", nos dará la opción de seleccionar el directorio donde se almacenarán los archivos temporales descargados para la instalación. Podemos dejar el que aparece por defecto.



Si pulsamos "*Siguiente*", aparecerá la opción de elegir la conexión de Internet. Podemos dejar la opción por defecto "*Use System Proxy Settings*" o "*Direct Connection*":



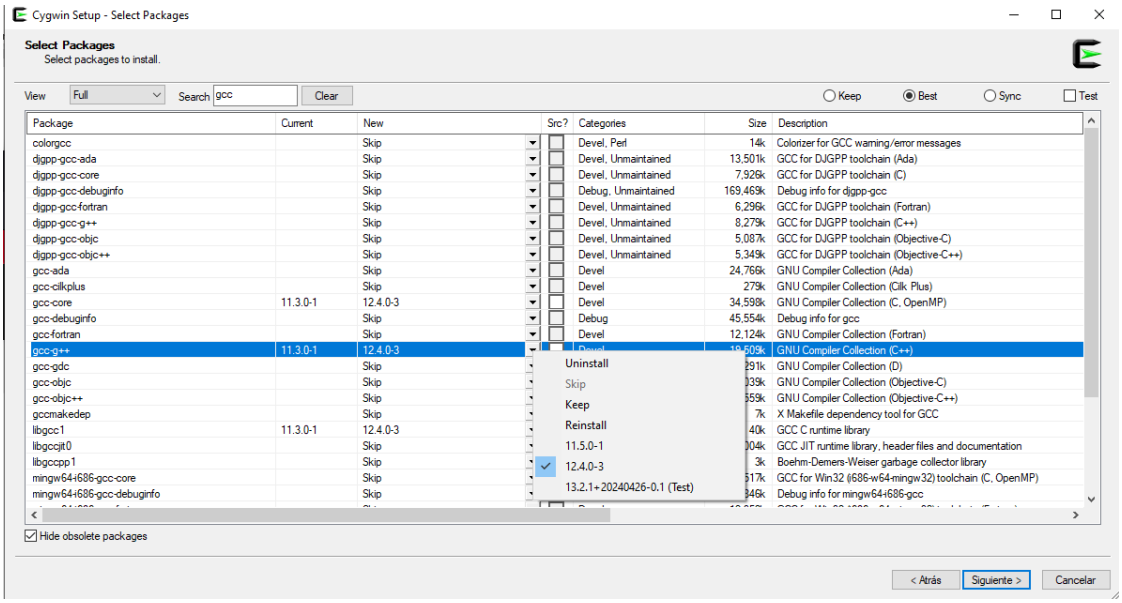
Al pulsar "*Siguiente*" nos permitirá seleccionar el sitio desde donde descargar los archivos necesarios, al ser mirrors todos los archivos serán iguales, por lo que podemos seleccionar el que queramos:



A continuación, nos aparecerá una lista con todos los paquetes disponibles. Es necesario cambiar la vista "*View*" a "*Full*" para poder ver todos los paquetes. Aquí será necesario seleccionar los siguientes paquetes:

- gcc-g++
- make
- gdb

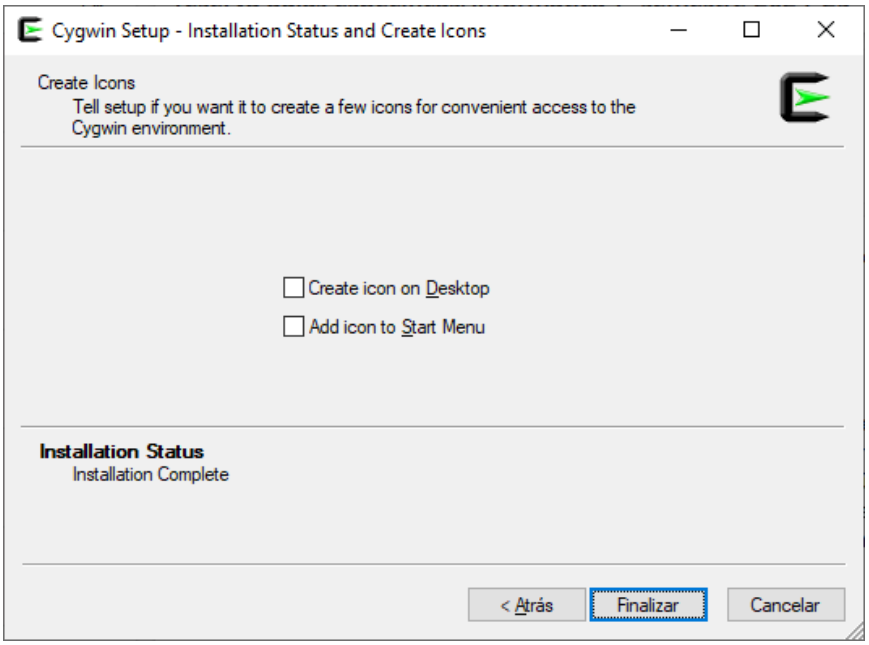
Para instalar los paquetes, una vez localizados en la lista debemos seleccionar la versión que queramos en la columna "*New*":



Las últimas versiones estables a la hora de hacer esta guía son:

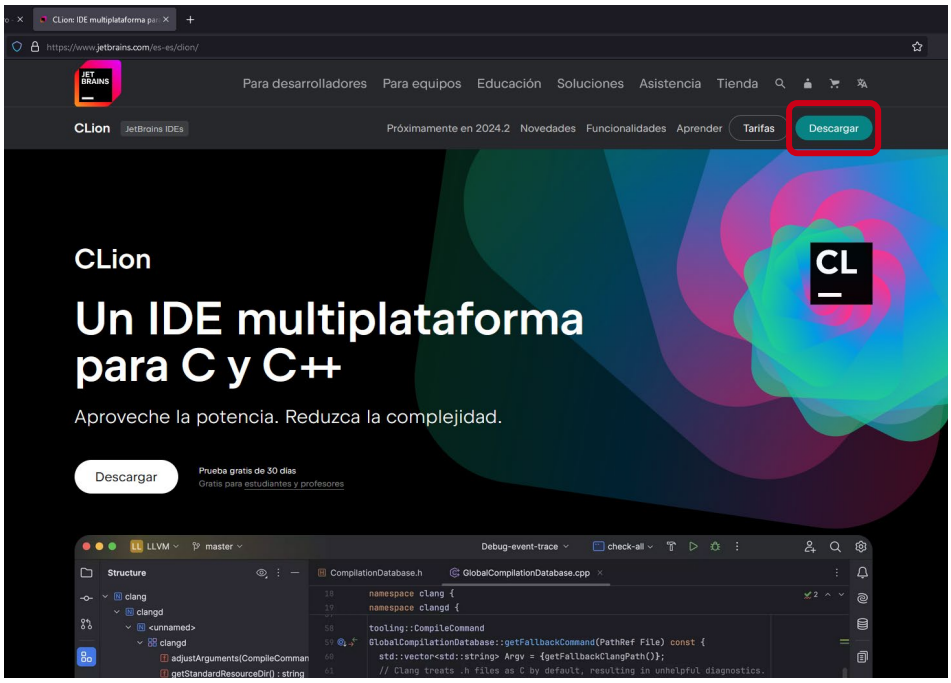
- Versión 12.4.0-3 de gcc-g++.
- Versión 4.4.1-2 de make.
- Versión 13.2-1 de gdb.

Cuando hayamos seleccionado los tres paquetes, pulsamos "*Siguiente*", apareciendo una nueva pantalla con los paquetes seleccionados (junto a otros paquetes necesarios para la instalación de los primeros). Si pulsamos de nuevo "*Siguiente*" comenzará la instalación, la cual puede tardar unos minutos. Cuando finalice aparecerá una nueva pantalla indicando que la instalación ha sido completada y permitiéndonos crear iconos. Pulsamos "*Finalizar*" para terminar la instalación.

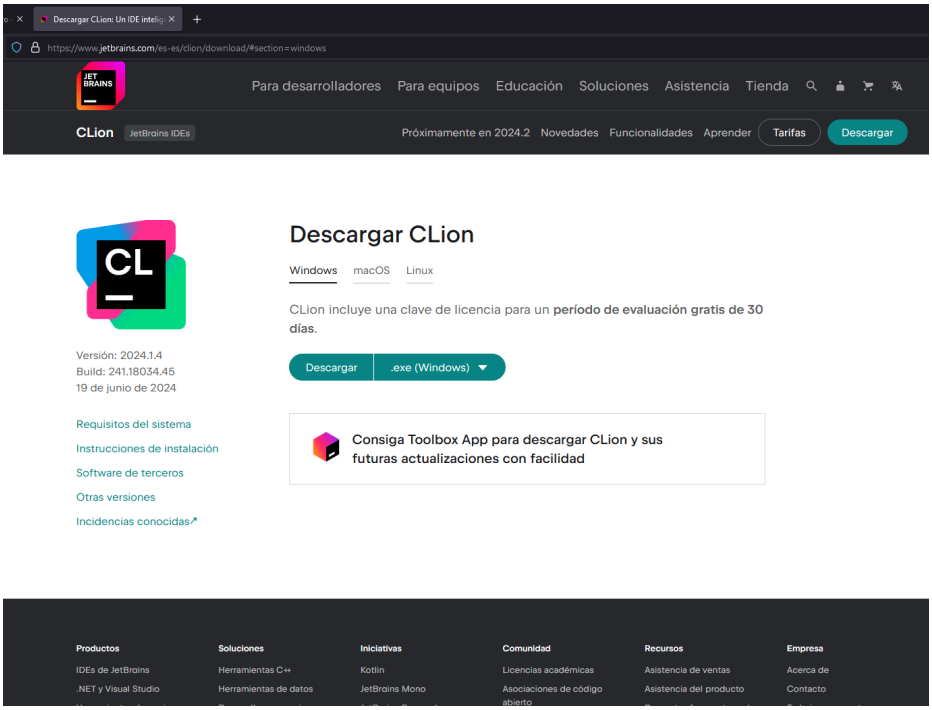


2. Instalación de CLion

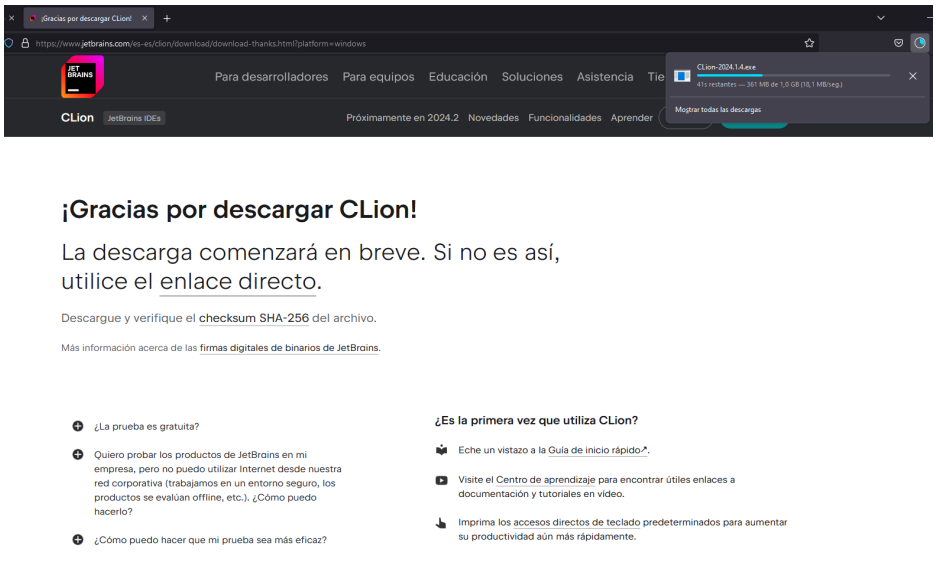
Una vez instaladas las herramientas necesarias para el funcionamiento de CLion es necesario descargar e instalar el entorno. Para ello, accedemos a su página web (https://www.jetbrains.com/es-es/clion/), y donde aparecerá una página como la mostrada a continuación:



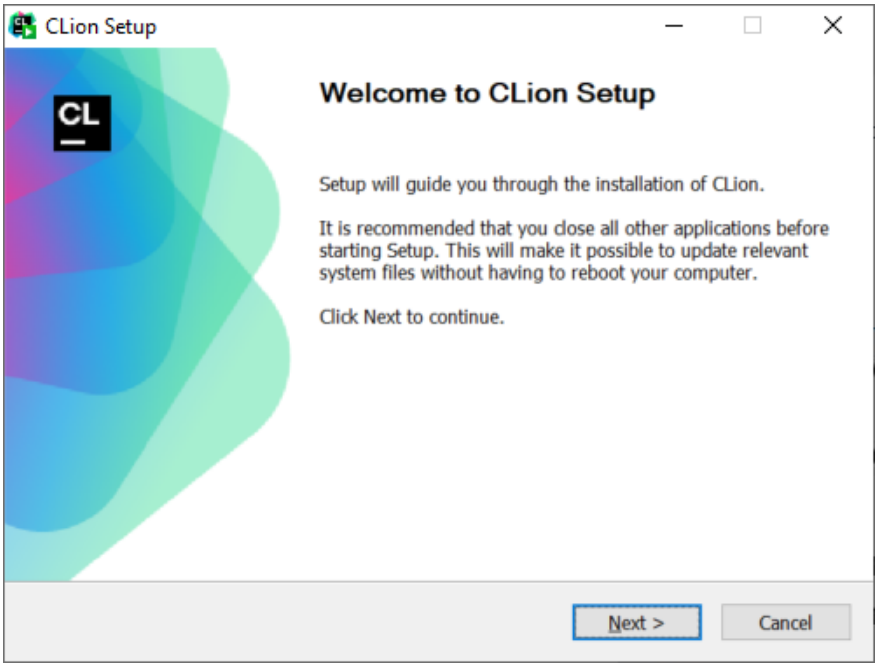
Seleccionamos la opción "*Descargar*", la cual nos llevará a la siguiente página:



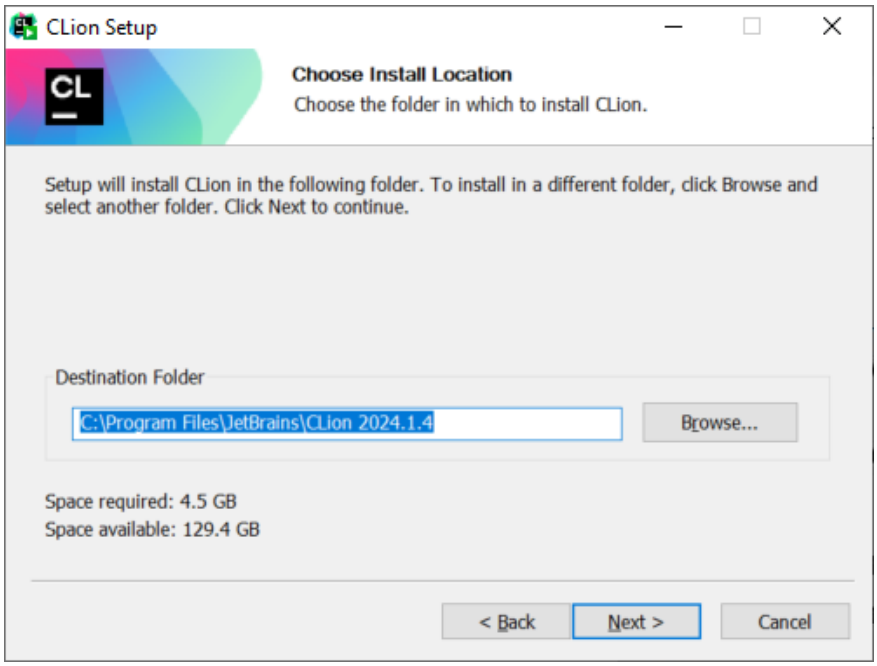
Comprobamos que la plataforma coincide con la descripción del equipo donde vamos a instalarlo y seleccionamos "*Descargar*", apareciendo la siguiente página:



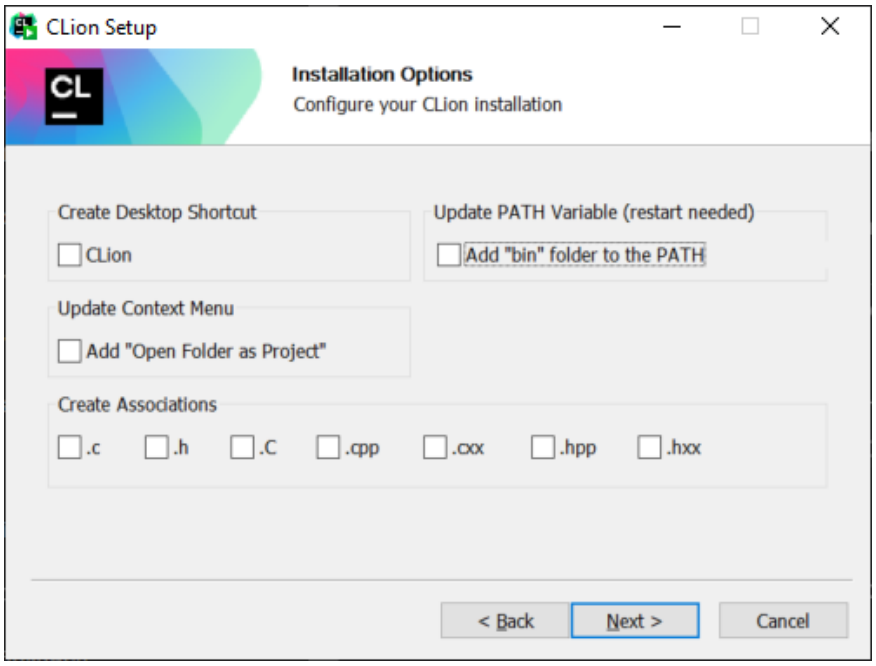
Una vez descargado el archivo, deberemos ejecutarlo, apareciendo la siguiente interfaz:



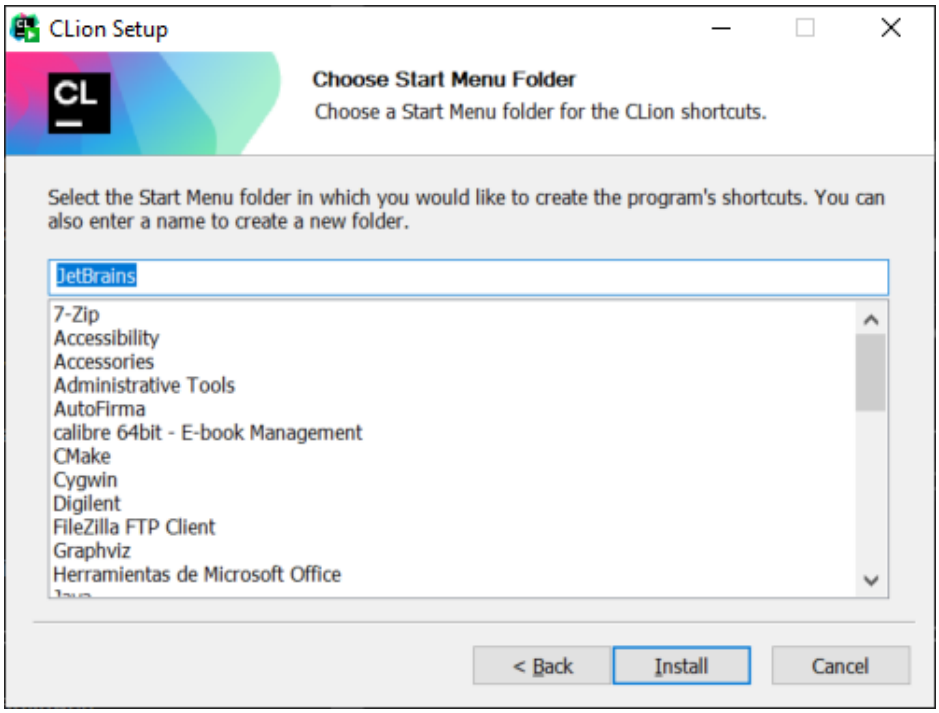
Pulsando "*Next*" nos permitirá seleccionar el directorio de instalación. Podemos dejar el de por defecto:



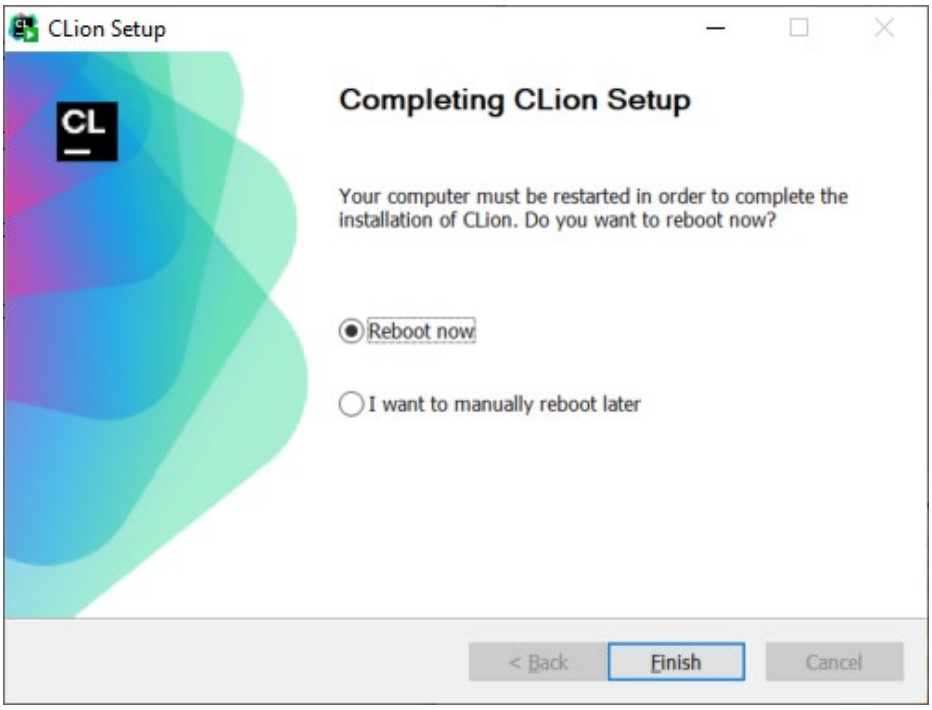
Una vez hayamos seleccionado la ruta, volvemos a pulsar "*Next*", pasando a la siguiente pantalla, donde nos permitirá seleccionar diferentes opciones:



Seleccionamos las opciones deseadas (en mi caso, "*Create Desktop Shorcut*" y "*Update PATH Variable*" y pulsamos "*Next*". Si hemos seleccionado crear un acceso directo, aparecerá la siguiente ventana:

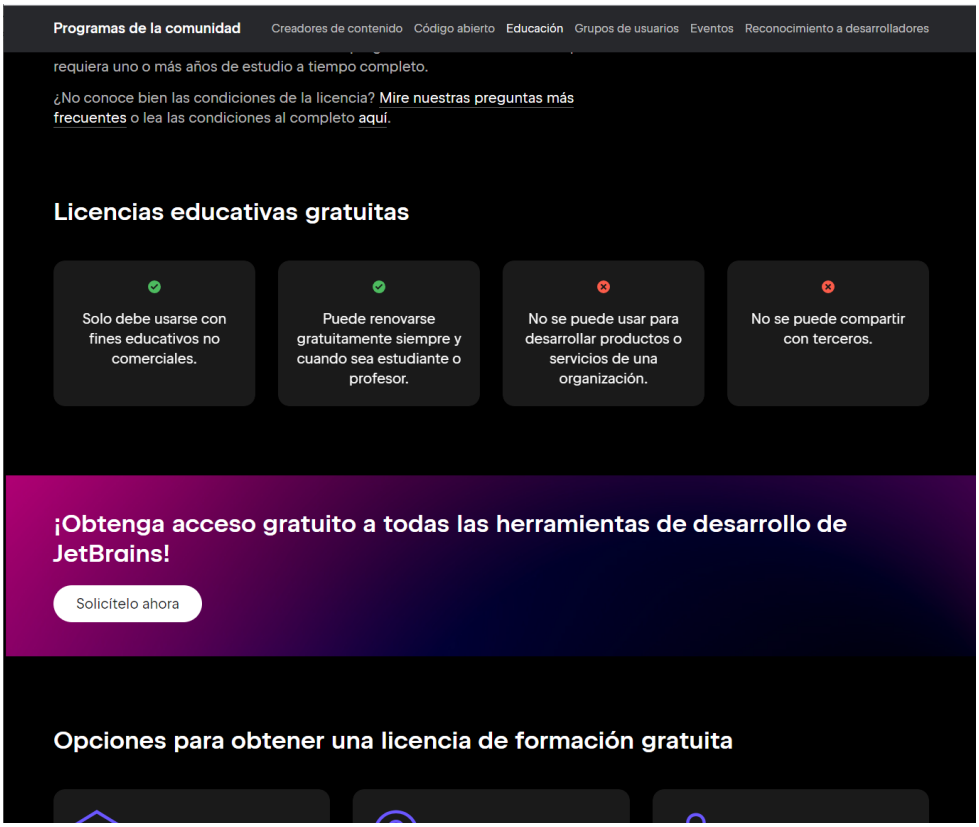


Seleccionamos "*Install*" y esperamos a que termine la instalación. Cuando termine, si hemos marcado la opción "*Update PATH Variable*" nos pedirá que reiniciemos el ordenador:

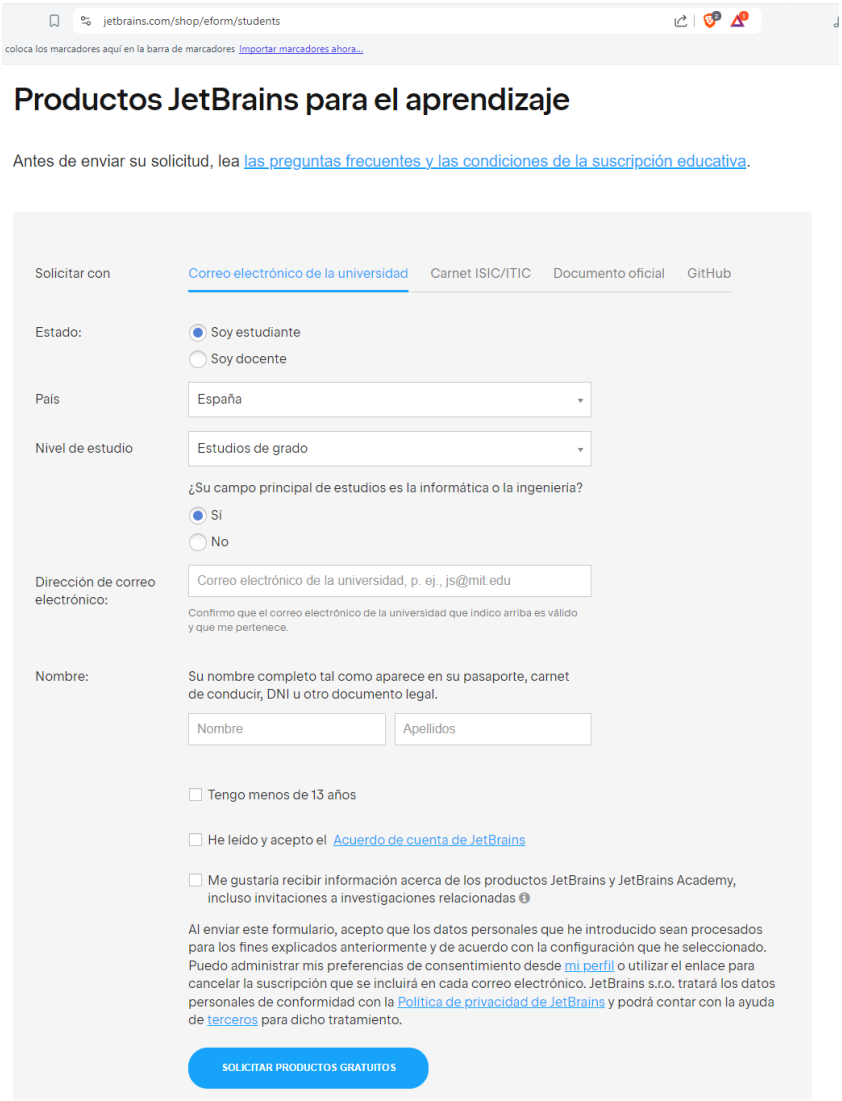


3. Licencia de CLion

Para utilizar CLion es necesario disponer de una licencia válida. Sin embargo, es posible adquirir una licencia gratuita para estudiantes. Para ello podemos ir a la dirección de licencias educativas: https://www.jetbrains.com/es-es/community/education/#students, donde aparecerá la siguiente página, y selección "*Solicítelo ahora*":



Tras seleccionar la opción, se nos abrirá la siguiente página, la cual nos permitirá seleccionar una opción para identificarnos como estudiantes (correo electrónico de la universidad, carnet ISIC/ITIC, documento oficial o GitHub). La manera más sencilla es utilizar el correo electrónico corporativo de la universidad (nombre@alumnos.urjc.es):

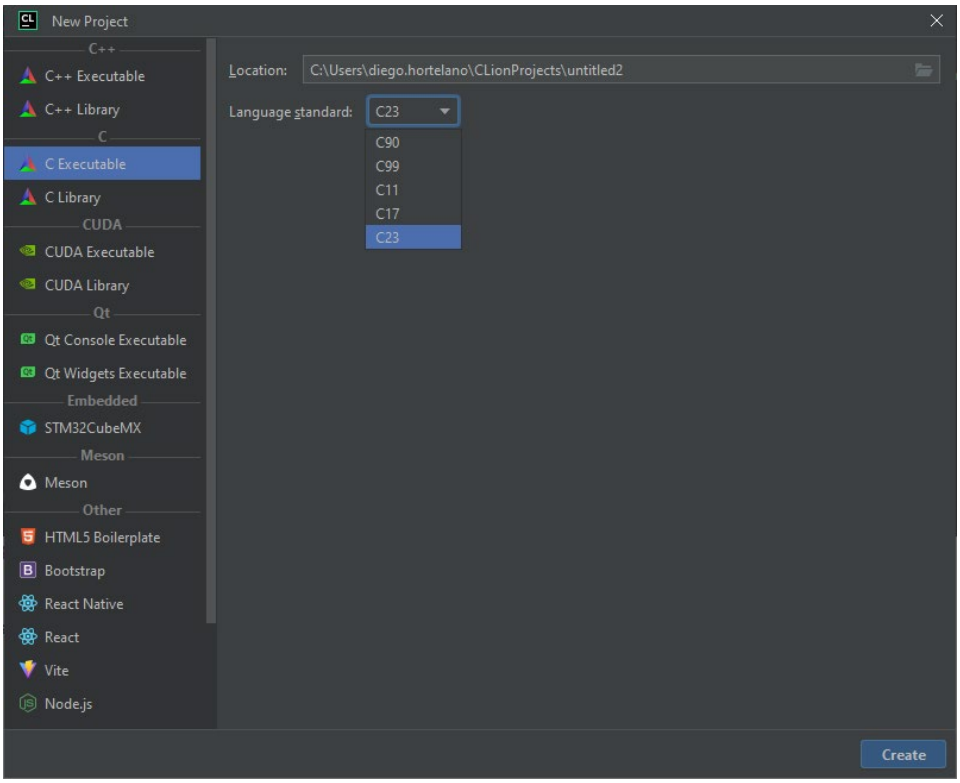


Tras seleccionar las opciones pertinentes y poner nuestro nombre y correo se nos creará una cuenta con una licencia de estudiante anual. Esta licencia puede renovarse anualmente mientras seamos estudiantes.

4. Creación de un nuevo proyecto en CLion

Cuando abramos por primera vez CLion es posible que nos pida una licencia válida. Si hemos completado el paso anterior será suficiente con iniciar sesión en la cuenta de JetBrains creada anteriormente para nuestra licencia de estudiante.

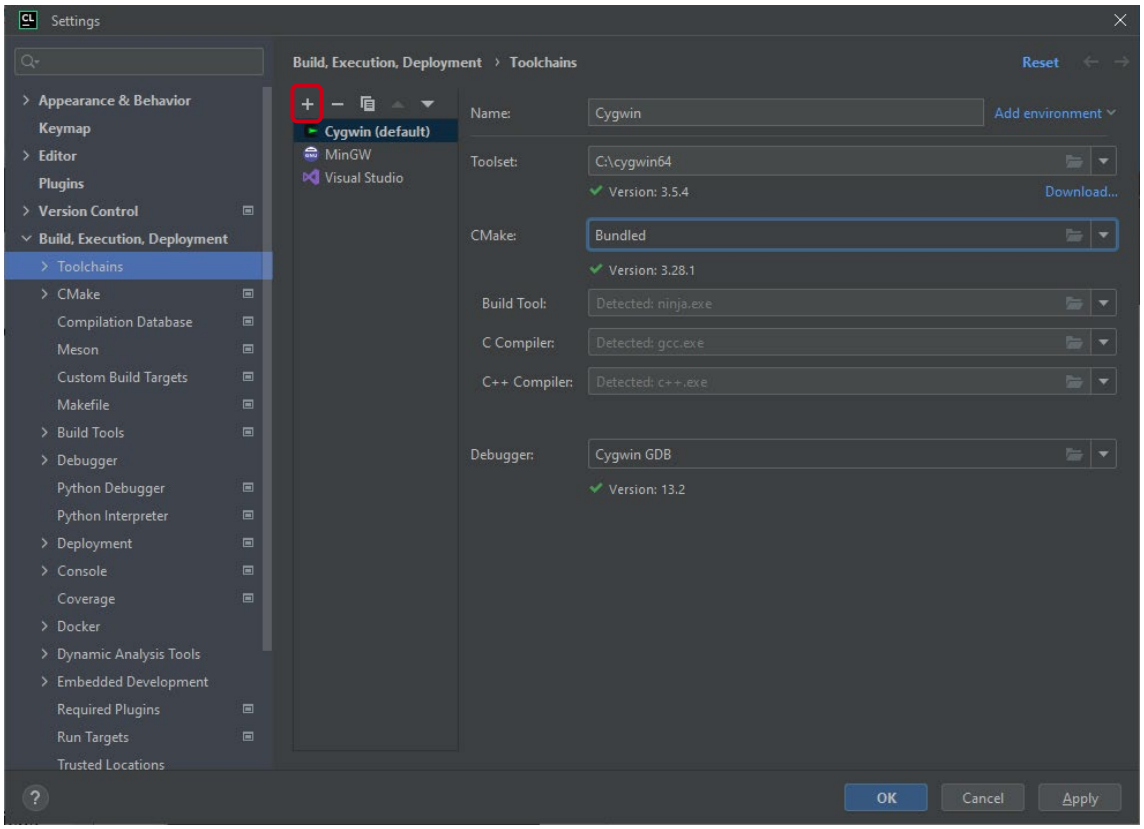
Para la creación de un nuevo proyecto debemos pulsar en "*File*" > "*New*" > "*Project*". El siguiente paso es seleccionar la opción "***C Executable***" del menú de la izquierda. Aquí nos aparecerá la siguiente ventana, donde nos permitirá seleccionar la ruta de nuestro proyecto y el estándar de C deseado.



Cuando tengamos todo preparado, seleccionamos Create. Es posible que nos aparezca un mensaje indicando si queremos abrir el nuevo proyecto en la ventana actual o en una nueva. Podemos seleccionar cualquier opción, ya que es posible volver a abrir un proyecto anterior, aunque cerremos la ventana actual.

5. Configuración del entorno

La primera vez que creemos un proyecto (el resto de los proyectos deben crearse con la misma configuración) es necesario revisar la configuración del mismo y modificarla si es necesario. Para ello, seleccionaremos "File" > "Settings", y en el menú de la izquierda de la pestaña que aparecerá, la pestaña "*Build, Execution, Deployment*" y "*Toolchains*". Si CLion ha detectado la instalación de Cygwin (o de MinGW si hemos optado por esta opción) deberá aparecer la configuración automáticamente, como se muestra en la siguiente captura. En caso contrario, debemos pulsar sobre el botón "+" y seleccionar Cygwin (o MinGW).

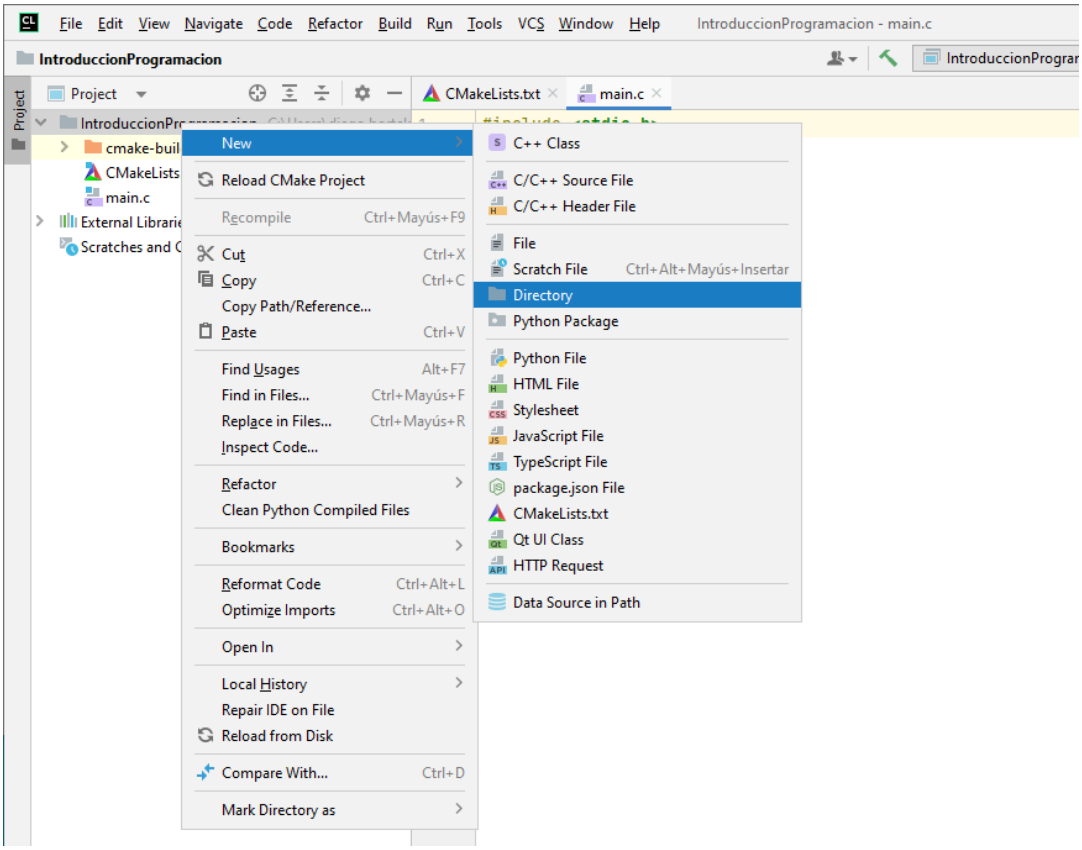


Si alguna opción no se configura automáticamente (en algunos casos la configuración de "*Debugger*" puede fallar), ésta aparecerá con una exclamación de error en rojo. En ese caso, debemos pulsar en el icono de la carpeta y buscar manualmente la ubicación del archivo. Este archivo *gbd.exe* correspondiente al *debugger* se encuentra en la carpeta de Cygwin ("*C:\cygwin64\bin\gbd.exe*" si hemos utilizado la ruta por defecto).

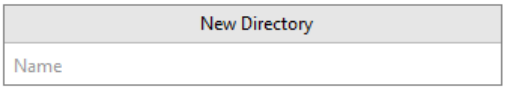
Una vez que todas las herramientas aparezcan correctamente reconocidas (con una marca de verificación verde ✓ bajo las mismas), habremos terminado de configurar nuestro entorno.

Anexo II: Uso de múltiples ficheros ejecutables en un mismo proyecto

Podemos tener diferentes ficheros ejecutables en un mismo proyecto, incluso utilizando diferentes directorios para separar nuestros ficheros (por ejemplo, para distinguir por temas). Para ello, podemos crear un directorio pulsando con el botón derecho sobre nuestro proyecto, y seleccionando la opción New – Directory.

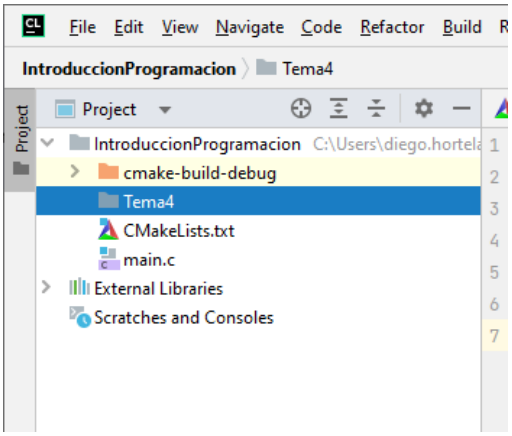


Nos aparecerá entonces una nueva ventana pidiendo el nombre del directorio a crear:

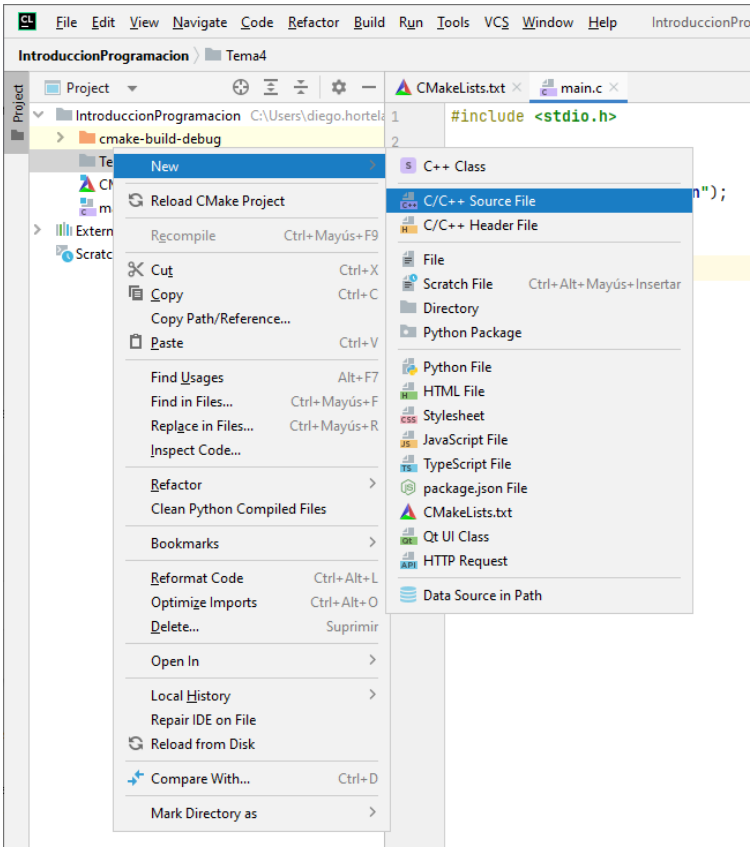


©2024 Diego Hortelano Haro, Gerardo Reyes Salgado, Manuel Rubio Sánchez. Algunos derechos reservados. Este documento se distribuye bajo la licencia "Atribución/Reconocimiento-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en https://creativecommons.org/licenses/by-sa/4.0/deed.es.

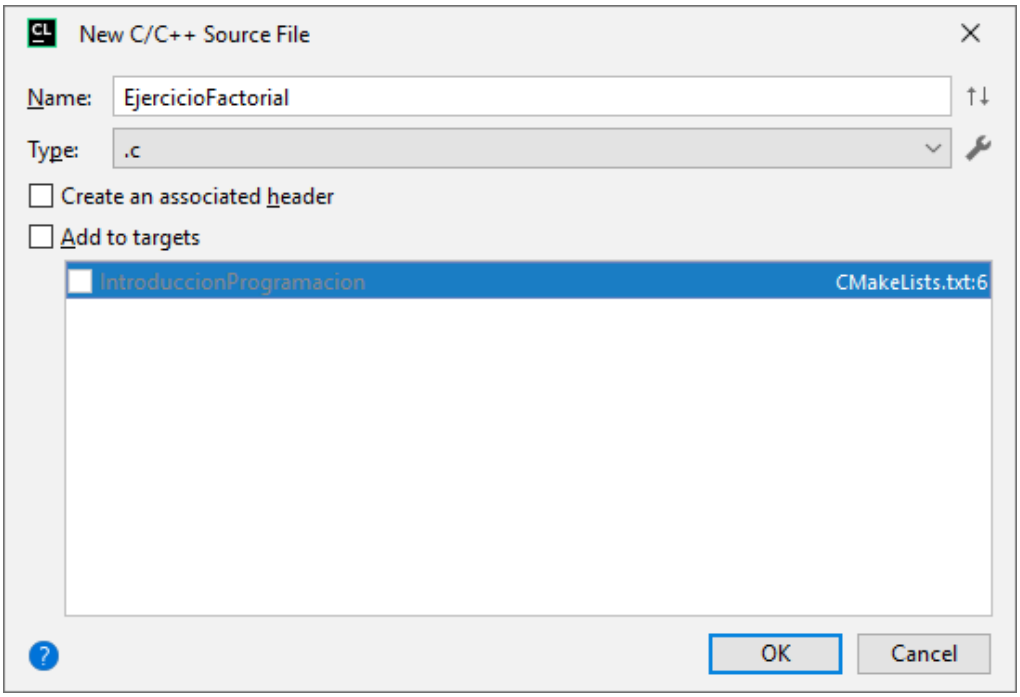
Al asignar un nombre y pulsar intro el directorio aparecerá en el proyecto:



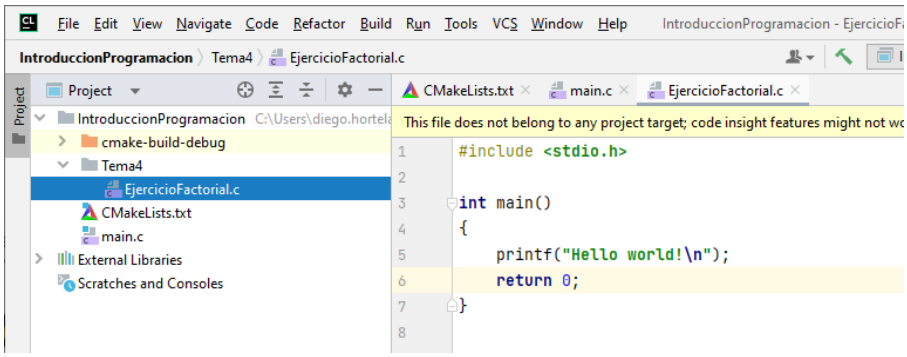
Pulsando con el botón derecho sobre este directorio y seleccionando la opción New – C/C++ Source File podremos crear un nuevo fichero C en nuestro directorio:



En ese momento se nos abrirá una nueva ventana, donde podemos asignar un nombre a nuestro fichero y seleccionar el tipo (importante que sea .c). Desmarcamos la opción de Add to targets y pulsamos OK.



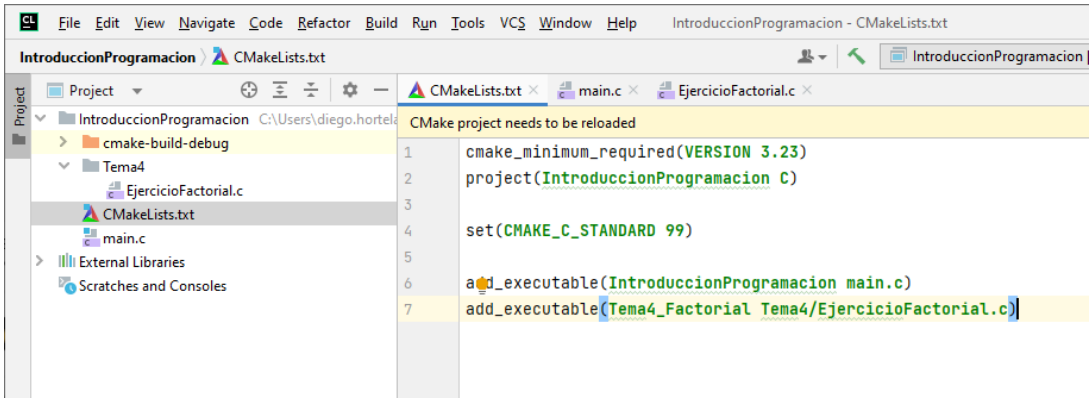
En ese momento aparecerá nuestro fichero dentro del directorio, como se muestra a continuación:



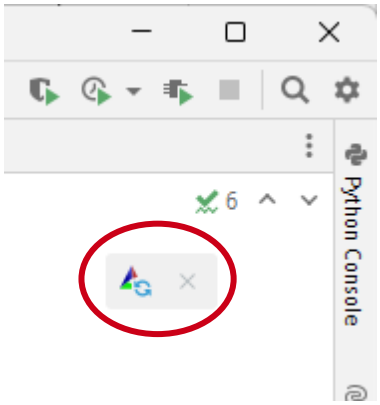
Sin embargo, antes de poder ejecutarlo, debemos modificar el archivo CMakeLists.txt, añadiendo una nueva línea por cada fichero ejecutable. En esta línea aparecerá el target del compilador, pudiendo asignar un nombre para su ejecución, y el nombre del fichero junto con su directorio, de la siguiente manera:

```
add_executable(nombre_ejecución directorio/fichero.c
```

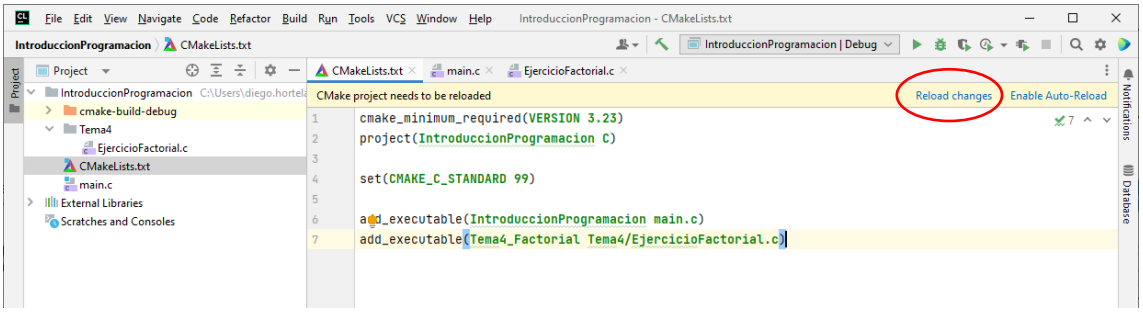
En caso de no haber utilizado un directorio podemos indicar directamente el nombre del fichero. En nuestro caso hemos definido el nombre de la configuración de ejecución "Tema4_Factorial", y el nombre del fichero junto con su directorio: "Tema4/EjercicioFactorial.c":



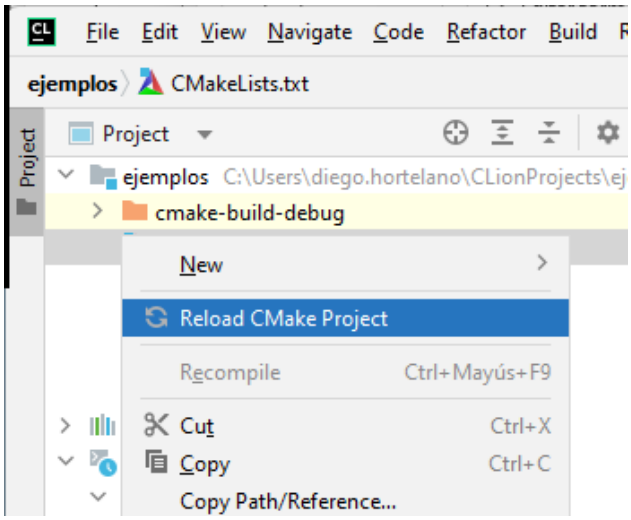
Tras ello es necesario actualizar la configuración del proyecto, siendo la manera más sencilla pulsando el icono de Load CMake Changes que aparece en la esquina superior derecha del entorno:



En otras versiones de CLion, el mensaje aparece como "Reload changes" en la sugerencia sobre el fichero:



También es posible actualizar esta configuración si hacemos click con el botón derecho del ratón sobre el archivo CMakeList.txt en la vista del proyecto y seleccionamos la opción Reload CMake Project:



Ahora podremos seleccionar la configuración de nuestro proyecto en el menú superior para que cuando presionemos el botón ejecutar, el compilador busque y ejecute el fichero ejecutable definido en la configuración: