# ÍNDICE

El código C++ usado para ilustrar las explicaciones teóricas, así como los programas propuestos como solución a los ejercicios de autocomprobación, están disponibles en la URL siguiente:
**https://canal.uned.es/series/magic/4xv87gl908w0w4wwsskc8s4sswcwscs**

# Tema 1

# FUNDAMENTOS DE PROGRAMACIÓN

## OBJETIVOS DOCENTES

Una vez estudiado el contenido del tema y realizados los ejercicios prácticos, debería saber:

- Discutir las ideas generales que han guiado la evolución de los lenguajes de programación.
- Discutir qué influencia ha tenido la arquitectura de la máquina de Von Neumann en los primeros lenguajes de programación imperativos.
- Discutir cómo se produce la ejecución de un programa en la máquina de Von Neumann.
- Discutir qué características tienen los lenguajes de bajo nivel (lenguaje máquina y ensamblador). Discutir las diferencias con los lenguajes de alto nivel y las ventajas de éstos frente a aquéllos.
- Discutir qué necesidades motivaron el desarrollo de los lenguajes de programación siguientes: FORTRAN, ALGOL, LISP, COBOL, BASIC, Visual BASIC, Prolog, SIMULA 67, Pascal, C, Modula-2, Ada, Smalltalk, C++, Java, sh, awk, Perl, JavaScript, PHP y Python. Discutir las principales características de estos lenguajes y qué nuevas capacidades aportó cada uno de ellos.
- Discutir en qué consiste la metodología de la programación estructurada.
- Discutir las características básicas de los cuatro paradigmas de programación fundamentales: programación imperativa, funcional, lógica y orientada a objetos.
- Discutir las características, y las ventajas y desventajas, de cada uno de los tres métodos siguientes de implementación de lenguajes de alto nivel: interpretación pura, compilación, y sistema híbrido de interpretación y compilación.
- Discutir en qué consiste el preprocesado del programa.

## 1.1. INTRODUCCIÓN

En este primer tema, que tiene carácter introductorio, se abordan algunos aspectos fundamentales relacionados con los lenguajes de programación. En §1.2/26, se describe brevemente la génesis, evolución y características distintivas de los lenguajes de programación más relevantes, destacando aquellas capacidades de cada lenguaje que han influido en el posterior desarrollo de otros lenguajes. En esta descripción ha sido preciso introducir gran cantidad de conceptos, que serán explicados en otros temas de esta Unidad Didáctica. Por tanto, en este punto no es importante entender completamente todos los detalles, sino comprender las ideas generales que han guiado la evolución de los principales lenguajes de programación.

En §1.3/56, se explican las características más relevantes y las diferencias entre los paradigmas de programación más comúnmente empleados, como son los paradigmas de la *programación imperativa*, *funcional*, *lógica* y *orientada a objetos*. El paradigma de programación soportado por el lenguaje condiciona decisivamente las capacidades del mismo. Por ello, los lenguajes de programación son comúnmente clasificados atendiendo a las facilidades que proporcionan para la aplicación de uno o varios de estos paradigmas.

- Los *lenguajes imperativos*, tales como FORTRAN, ALGOL, COBOL, SIMULA 67, Pascal, C, Modula-2 y Ada, facilitan la aplicación del paradigma de la *programación imperativa*. A su vez, los lenguajes imperativos son clasificados en aquellos que facilitan la *programación estructurada* y aquellos que no la facilitan.

- Los *lenguajes funcionales*, como LISP, COMMON LISP y Scheme, facilitan la aplicación del paradigma de la *programación funcional*.

- Los *lenguajes lógicos*, como Prolog, han sido ideados para facilitar la *programación lógica*.

- Los *lenguajes orientados a objetos*, como Smalltalk, C++, Java y Python soportan el paradigma de la *programación orientada a objetos*. Dado que la programación orientada a objetos evolucionó a partir de la programación imperativa, los lenguajes orientados a objetos también facilitan la programación imperativa.

Finalmente, los tres mecanismos existentes para el procesamiento del programa son descritos en §1.4/61. Estos métodos de implementación son la *interpretación pura*, la *compilación* y un *sistema híbrido* de ambos.

## 1.2. EVOLUCIÓN DE LOS LENGUAJES DE PROGRAMACIÓN

El progresivo desarrollo y abaratamiento del hardware de computación ha tenido una influencia fundamental en el desarrollo de los lenguajes de programación.

Por una parte, el impresionante avance en las capacidades de computación ha permitido desarrollar programas cada vez más complejos, surgiendo la necesidad de desarrollar metodologías de programación adecuadas para el desarrollo de programas de grandes dimensiones y complejidad, y lenguajes de programación que soporten dichos desarrollos.

Por otra parte, el abaratamiento del hardware ha hecho que el tiempo de trabajo del programador sea una contribución cada vez más importante al coste del desarrollo de los sistemas informáticos. En la década de 1950, el coste de los ordenadores era muy superior al coste del trabajo de los programadores. Este hecho guió el desarrollo de los lenguajes de programación en esa época. Por ejemplo, los recursos para la descripción del programa que proporcionaba la primera versión de FORTRAN estaban muy cercanos al lenguaje máquina. Esto limitaba enormemente la expresividad del lenguaje, si bien permitía que los programas escritos en FORTRAN fueran traducidos por el compilador de FORTRAN a programas en código máquina muy eficientes.

A medida que el coste del hardware fue reduciéndose, la repercusión que los costes debidos al desarrollo y depurado del programa tenían en el coste total del proyecto fue haciéndose cada vez mayor. Este hecho motivó, en la década de 1970, el planteamiento de una nueva metodología de programación, llamada *programación estructurada*, y de nuevos lenguajes que facilitaban ponerla en práctica. El objetivo de esta metodología era facilitar el desarrollo de programas fácilmente legibles por el programador, y consecuentemente más sencillos de entender, desarrollar y depurar.

Este mismo objetivo dio lugar a la aparición, en la década de 1980, de la metodología de la *programación orientada a objetos*. En este caso, el propósito fue también facilitar el diseño y mantenimiento del código, así como su reutilización.

El índice de la Comunidad de Programación TIOBE es un indicador de la popularidad de los lenguajes de programación. El cálculo de este índice se actualiza una vez al mes (`https://www.tiobe.com/tiobe-index/`). TIOBE es una compañía que se especializa en determinar y realizar el seguimiento de los mejores lenguajes de programación. Para realizar esta tarea, la compañía se basa en parámetros como: consultas realizadas en buscadores como Google, Yahoo y MSN, cantidad de ingenieros expertos en un determinado lenguaje, aplicaciones

desarrolladas en ese lenguaje, etc. La página oficial de TIOBE es: `www.tiobe.com`. Según los cálculos de octubre de 2020, los cinco lenguajes más populares son, de más a menos popularidad: C, Java, Python, C++ y C#.

La evolución de este índice en el tiempo muestra que los lenguajes más populares van cambiando. Por ello, cualquier desarrollador de software ha de estar preparado para aprender diferentes lenguajes. Resulta así esencial para un buen programador conocer el vocabulario y los conceptos fundamentales de los lenguajes de programación en general de modo que pueda aprender fácilmente nuevos lenguajes de programación, así como evaluar cuál es el mejor lenguaje para realizar un desarrollo en concreto.

A continuación, se explica la génesis y evolución de algunos de los lenguajes de programación más relevantes, tanto desde el punto de vista conceptual, como desde el punto de vista de la gran popularidad que alcanzaron en su día. La fecha de aparición de algunos de estos lenguajes se indica en la Figura 1.1, señalándose mediante flechas la relación existente entre ellos.

### 1.2.1. La máquina de Von Neumann

Antes de embarcarnos en el estudio de los lenguajes de programación, es necesario comprender algunos conceptos básicos del diseño de los ordenadores, ya que la arquitectura de los computadores ha condicionado de manera decisiva el desarrollo de los lenguajes de programación.

La *máquina de Von Neumann*, que fue diseñada a finales de los años 1940 en el Institute for Advanced Study (Princeton, EE.UU.), puede considerarse la precursora de los ordenadores actuales. En efecto, tal es la relación existente entre los ordenadores actuales y aquella máquina, que la arquitectura de los ordenadores actuales se denomina *arquitectura de Von Neumann*.

En la Figura 1.2 se ha representado la arquitectura básica de la *máquina de Von Neumann*. En la *memoria* se almacenan tanto las instrucciones que componen el programa a ejecutar en la máquina, como los datos. Ambos, las instrucciones y los datos, son codificados mediante palabras binarias. Una *palabra binaria* es una secuencia, de una determinada longitud, de unos y ceros.

La memoria está formada por una secuencia de posiciones de almacenamiento, que están numeradas consecutivamente. El número correspondiente a una posición de memoria se denomina la *dirección* de la posición de memoria. Cada posición de almacenamiento puede alojar una palabra binaria.

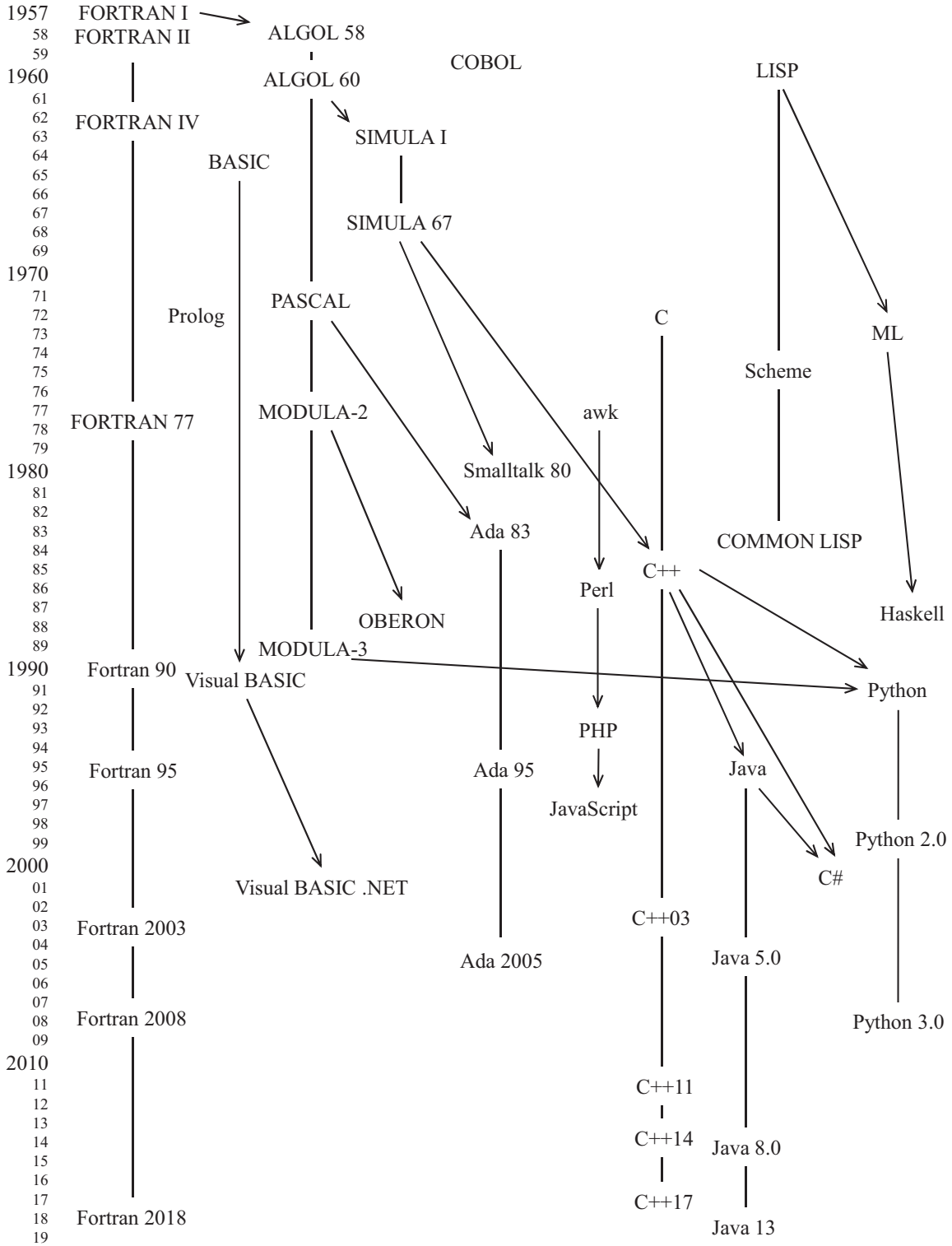


**Figura 1.1:** Genealogía de algunos de los lenguajes de programación más relevantes.

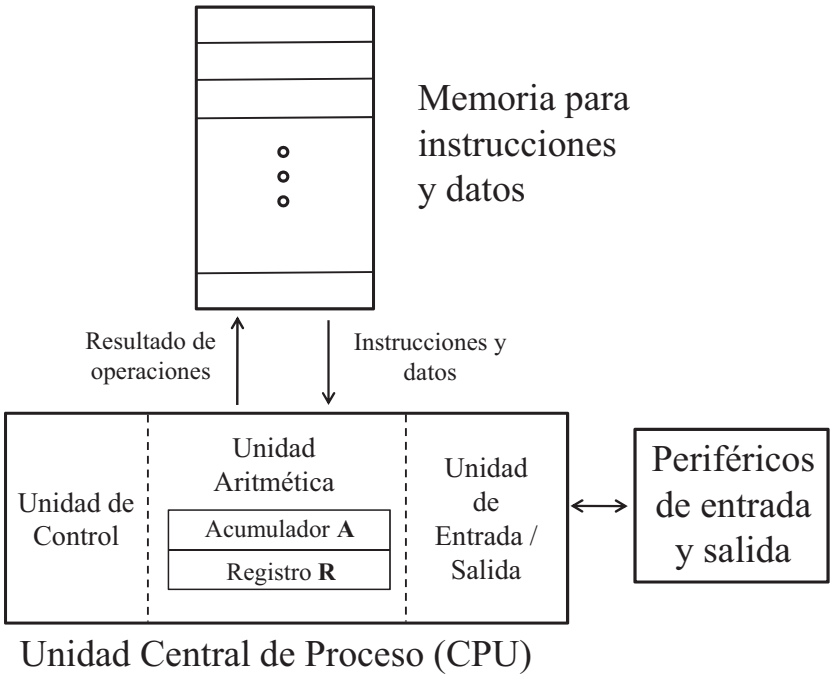


**Figura 1.2:** Organización de la máquina de Von Neumann.

Se puede acceder a la palabra binaria contenida en una posición de memoria especificando la dirección de la posición de memoria. El procedimiento de acceso es independiente de si lo que contiene la posición de memoria es un dato o una instrucción.

La *unidad central de proceso* (abreviado CPU, del inglés Central Processing Unit) está básicamente compuesta de tres componentes: la *unidad de control*, la *unidad aritmética* y la *unidad de entrada/salida*..

La *unidad de control* lee de memoria la instrucción del programa y genera las señales de control necesarias para que se realicen las tareas descritas en el programa.

La CPU tiene un registro especial, llamado *contador de programa*, donde se almacena la dirección de memoria en la cual se encuentra la siguiente instrucción a ejecutar del programa.

En la *unidad aritmética* se realizan las operaciones aritméticas. La unidad aritmética recibe las señales de control de la unidad de control, así como los datos provenientes de la memoria, en aquellos casos en que alguno de los operandos esté almacenado en memoria. La unidad aritmética contiene dos registros, que se emplean para almacenar los operandos y resultados de las operaciones aritméticas. Uno de estos registros recibe el nombre de *acumulador*.

Obsérvese que el contenido de una dirección de memoria es interpretado como un dato o como una instrucción dependiendo de qué parte de la máquina lo reciba. Las palabras binarias almacenadas en memoria que son leídas por la unidad de control, son interpretadas por la unidad de control como instrucciones. Las palabras leídas por la unidad aritmética son interpretadas como datos por la unidad aritmética.

La *unidad de entrada/salida* permite introducir en la máquina tanto los datos como las instrucciones del programa. También permite comunicar al mundo exterior los resultados de la ejecución del programa.

Un programa en ejecución en la máquina de Von Neumann es una secuencia de palabras binarias, las cuales describen las acciones que debe realizar el programa. Esta codificación mediante palabras binarias, que sigue una sintaxis dependiente de la CPU de la máquina, se denomina *código máquina* o *lenguaje máquina*.

La memoria de la máquina de Von Neumann contiene 1024 palabras binarias, cada una con una longitud de 40 bits (un bit es un dígito con valor 0 o 1).

- Cuando la palabra binaria se interpreta como dato, ésta representaba un número entero con signo codificado en base 2.
- Cuando la palabra binaria se interpreta como instrucción de programa, representa dos instrucciones: una codificada mediante los 20 bits más significativos de la palabra (los situados más a la izquierda) y la otra mediante los 20 bits menos significativos de la palabra (los situados más a la derecha).

Entre las instrucciones soportadas por la máquina de Von Neumann, cabe destacar aquellas que describen los siguientes tipos de operaciones:

- *Operaciones aritméticas*. La unidad aritmética puede realizar las operaciones suma, resta, multiplicación y división, y también calcular el valor absoluto de un número. La instrucción que describe una operación aritmética no sólo indica la operación a realizar, sino también la ubicación de los operandos. Generalmente se asume que uno de los operandos está almacenado en el registro acumulador y el otro en una posición de memoria, cuya dirección es preciso indicar. El resultado de la operación se almacena en el registro acumulador.
- *Movimiento de datos*. Este tipo de instrucciones hacen que el contenido de una posición de memoria se copie a uno de los registros, o viceversa, o que el contenido de uno de los registros de la CPU se copie al otro. Cuando el dato se mueve desde o hacia una posición de memoria, en la instrucción debe especificarse cuál es la dirección de dicha posición de memoria.

– *Control del flujo del programa*. Las instrucciones del programa se van ejecutando secuencialmente, una tras otra, en el mismo orden en que están almacenadas en memoria. La instrucción *goto* permite describir saltos en el flujo del programa. Hay dos modalidades de sentencia *goto*.

1. En la primera, denominada *goto incondicional*, el salto se produce siempre.
2. En la segunda, denominada *goto condicional*, sólo se produce el salto si se satisface una condición: que el número que se encuentre almacenado en ese momento en el acumulador sea mayor o igual que cero.

En ambos casos, en la sentencia *goto* debe especificarse la dirección de memoria a la que se salta y, dentro de ésta, si se salta a la instrucción codificada por los 20 bits más significativos o por los 20 bits menos significativos (recuérdese que cada palabra en memoria tiene 40 bits, que se interpretan como dos instrucciones de 20 bits).

Obsérvese que cada una de las instrucciones del lenguaje máquina realiza una única tarea simple. Además, cada instrucción está autocontenida. Es decir, contiene toda la información necesaria para ser ejecutada. Asimismo, obsérvese que la información que contiene una instrucción en lenguaje máquina es básicamente la siguiente: cuál es la operación a realizar, qué operandos participan en la operación, dónde debe almacenarse el resultado de la operación y dónde está situada la siguiente instrucción.

La ejecución, en la máquina de Von Neumann, de un programa en código máquina se realiza básicamente aplicando el algoritmo del Código 1.1

```
1  Inicializar el contador de programa (PC)
2  repeat
3     Llevar de la memoria a la CPU la instrucción apuntada por el PC
4     Incrementar el PC, para que apunte a la instrucción siguiente
5     Decodificar la instrucción
6     Ejecutar la instrucción
7  end repeat
```

**Código 1.1:** Algoritmo para ejecución de programas en la máquina de Von Neumann.

El primer paso del algoritmo es cargar en el registro contador de programa (denominado abreviadamente PC, del inglés Program Counter) la dirección de memoria en que se encuentra almacenada la primera instrucción del programa. El cuerpo del bucle **repeat** se repite una y otra vez, hasta que se ejecuta una instrucción que

indica que el programa ha finalizado, momento en el cual termina la ejecución del programa.

La máquina tarda menos tiempo en ejecutar una operación en la unidad aritmética, que en mover una instrucción o dato entre la memoria y la CPU. Por este motivo, la velocidad de ejecución del programa está limitada fundamentalmente por la velocidad de transmisión de datos entre la memoria y la CPU.

La programación de la máquina de Von Neumann, así como de los primeros ordenadores que se desarrollaron posteriormente, a principios de los años 1950, consistía en escribir las instrucciones en código máquina que describían las operaciones que debía realizar el ordenador. Cada tipo de ordenador tenía su propio juego de instrucciones, diferente del de los demás, con lo cual los programas no eran portables de un ordenador a otro.

La programación en lenguaje máquina es una tarea muy tediosa. Dado que cada instrucción en código máquina es una palabra binaria (secuencia de unos y ceros), los programas en lenguaje máquina resultan difíciles de comprender para el programador.

Una dificultad añadida en la máquina de Von Neumann y en los primeros ordenadores, que hacía que los programas fueran difíciles de modificar, es que cuando en el programa se hacía referencia a una determinada posición de memoria, debía especificarse la dirección de dicha posición de memoria. Esto se conoce como *direccionamiento absoluto*.

Supongamos un programa en lenguaje máquina, escrito con direccionamiento absoluto, que se encuentra almacenado en memoria. Muchas de las instrucciones del programa hacen referencia a otras posiciones dentro del programa. Por ejemplo, para referirse a datos o para indicar a qué instrucciones deben producirse los saltos en las sentencias *goto*. Si se inserta una instrucción en el programa, deben modificarse todas las instrucciones del programa que hacen referencia a posiciones situadas a continuación del punto de inserción, dado que las instrucciones situadas a continuación del punto de inserción han debido desplazarse en la memoria a fin de alojar la instrucción insertada.

Un problema similar surgía cuando se eliminaba una instrucción del programa: todas las instrucciones que hacían referencia a instrucciones situadas a continuación de la instrucción borrada debían ser modificadas. Este último problema fue fácilmente solucionado, introduciendo en el lenguaje máquina una instrucción cuyo significado era “no operación” y que podía ser empleada para reemplazar a la instrucción borrada.

### 1.2.2. Lenguaje ensamblador

La programación en lenguaje máquina era muy tediosa y propensa a errores. Por ello, el desarrollo en la década de 1950 del *lenguaje ensamblador* supuso un avance muy importante. La novedad consistía en que en el lenguaje ensamblador se asigna un nombre a cada operación del código máquina. También, el lenguaje ensamblador permite asignar etiquetas a las posiciones de almacenamiento.

Aunque el lenguaje ensamblador es más fácilmente comprensible que el lenguaje máquina, sigue siendo difícilmente inteligible por el programador. Por ejemplo, se muestran a continuación tres instrucciones en lenguaje máquina.

```
00000010101111001010
00000010111111001000
00000011001110101000
```

La primera carga el contenido de una posición de memoria en un registro, la segunda suma al registro el contenido de una posición de memoria (el resultado queda almacenado en el registro) y la tercera copia el dato del registro a una posición de memoria. La tarea realizada por estas tres sentencias puede describirse en *lenguaje ensamblador* de la forma siguiente:

```
LOAD  I
ADD   J
STORE K
```

Un programa, denominado *ensamblador*, es el encargado de traducir automáticamente el código ensamblador a código máquina. El lenguaje ensamblador y el programa ensamblador, así como el lenguaje máquina, es específico a cada CPU, con lo cual el código de los programas escritos en ensamblador no es portable de una CPU a otra.

El lenguaje máquina y el lenguaje ensamblador se conocen como *lenguajes de bajo nivel*, ya que se encuentran muy próximos a la máquina. Las tres instrucciones anteriores en lenguaje ensamblador equivaldrían a la siguiente sentencia de un programa escrito empleando un *lenguaje de alto nivel*

```
K=I+J
```

la cual indica que el dato almacenado en la variable `I` debe sumarse al dato almacenado en la variable `J` y el resultado debe almacenarse en la variable `K`.

### 1.2.3. FORTRAN

FORTRAN es uno de los primeros lenguajes de programación de alto nivel. La motivación principal para el desarrollo del lenguaje FORTRAN fue el alto coste que tenía codificar los programas empleando lenguaje máquina o ensamblador. Una contribución importante a este coste era el tiempo que debía dedicarse al *depurado* del código. Es decir, a encontrar y corregir los errores cometidos en los programas escritos en lenguaje ensamblador o máquina.

El lenguaje FORTRAN, que toma su nombre de *FORmula TRANslation*, fue desarrollado por un grupo de investigación de IBM liderado por John Backus. El objetivo que se perseguía con el desarrollo del lenguaje FORTRAN era permitir al programador especificar un procedimiento numérico empleando un lenguaje conciso, similar al de las matemáticas, y obtener automáticamente, a partir de esta especificación, un programa eficiente en código máquina que realice dicho procedimiento numérico.

El desarrollo de un lenguaje de programación no sólo implica la definición formal del mismo, sino también su *implementación*, entendiendo como tal el desarrollo de compiladores o intérpretes que traduzcan automáticamente los programas escritos en el lenguaje de programación al código máquina de la CPU del ordenador.

El objetivo fundamental en el desarrollo de FORTRAN fue diseñar un compilador que generara código máquina optimizado. La mayor parte del esfuerzo necesario para el desarrollo del lenguaje FORTRAN estuvo precisamente dedicado a resolver el problema de la traducción de los programas escritos en FORTRAN a código máquina eficiente. Esto es, estuvo dedicado al desarrollo del *compilador* de FORTRAN.

Debe tenerse en cuenta que en los años 1950 los ordenadores eran lentos y tenían una memoria con una capacidad muy reducida. Por otra parte, el coste de los ordenadores era muy grande comparado con el coste del trabajo de los programadores. A consecuencia de ello, si el código máquina generado por el compilador de FORTRAN contuviese un número de instrucciones máquina significativamente mayor que el que hubiera podido ser escrito directamente por un programador experto, entonces hubiera sido más rentable continuar escribiendo los programas en código máquina o ensamblador, en lugar de emplear el lenguaje FORTRAN.

El compilador de FORTRAN se desarrolló para un ordenador en concreto: el IBM 704, que se fabricó a partir de 1954. Este ordenador proporcionaba nuevas capacidades, que a su vez facilitaron el desarrollo del compilador. Entre estas nuevas capacidades cabe destacar que el ordenador IBM 704 disponía de hardware espe-

cífico para realizar operaciones entre números reales expresados en coma flotante. Obsérvese que los ordenadores desarrollados hasta ese momento tenían una unidad aritmética capaz de realizar operaciones entre números enteros, pero carecían de hardware específico para realizar operaciones entre números reales en coma flotante. En consecuencia, las operaciones entre reales en coma flotante debían ser simuladas mediante software, lo cual consumía un tiempo de computación muy considerable.

El periodo de desarrollo del lenguaje FORTRAN comenzó en enero de 1955 y continuó hasta que se lanzó la primera versión del compilador en abril de 1957. El lenguaje implementado se denominó FORTRAN I.

Las sentencias de control del flujo del programa de FORTRAN I se basaron en las instrucciones máquina del IBM 704, de modo que la traducción de la sentencia FORTRAN a la instrucción máquina fuera óptima.

El ordenador IBM 704 proporcionaba una instrucción máquina de salto condicional con tres opciones. La instrucción comparaba el valor almacenado en una posición de memoria con el valor almacenado en un registro. Dependiendo del resultado de la comparación (mayor, igual o menor), se realizaba el salto a la correspondiente posición de memoria de las tres especificadas en la instrucción: se saltaba a la posición especificada en primera, segunda o tercera posición, respectivamente.

La sentencia de bifurcación de FORTRAN I es análoga a la instrucción máquina de salto condicional:

```
IF (expresión_aritmética) N1, N2, N3
```

donde `N1`, `N2` y `N3` son etiquetas que señalan las sentencias del programa a las cuales debe darse el salto, dependiendo de que el resultado de evaluar la expresión aritmética sea negativo, cero o mayor que cero, respectivamente.

La sentencia iterativa de FORTRAN I, que permitía ejecutar un número determinado de veces cierto fragmento de código, también era directamente traducible al código máquina del ordenador IBM 704. La sentencia iterativa en FORTRAN I era de la forma:

```
DO N1 variable = primer_valor, ultimo_valor
```

donde `N1` es la etiqueta de la última sentencia del fragmento de código a repetir, y la sentencia escrita a continuación de esta sentencia `DO`, la primera. La sentencia iterativa ejecuta dicho fragmento de código, denominado *cuerpo del lazo*, cierto número de veces, que está determinado por la diferencia entre los valores `ultimo_valor` y `primer_valor`. La variable `variable` se denomina *variable del lazo*.

La ejecución de la sentencia iterativa se realiza de la forma siguiente. Se asigna a la variable del lazo el valor `primer_valor` y se ejecuta el cuerpo del lazo. Una vez ejecutado el cuerpo del lazo, se comprueba si el valor de la variable del lazo es igual a `ultimo_valor`, en cuyo caso finaliza la ejecución de la sentencia iterativa. En caso contrario, se incrementa la variable del lazo, se ejecuta el cuerpo del lazo y vuelve a compararse el valor de la variable con `ultimo_valor`. Así sucesivamente.

En FORTRAN I no se declara el tipo de las variables, sino que se empleaba la primera letra del nombre de la variable para identificar su tipo: aquellas variables cuyo nombre comenzaba por `I`, `J`, `K`, `L`, `M` o `N` eran consideradas de tipo entero, mientras que el resto eran consideradas de tipo real. El nombre de las variables podía tener hasta 6 caracteres.

FORTRAN I se convirtió pronto en un lenguaje muy ampliamente usado para la programación científica. Por una parte, proporciona una notación para las expresiones aritméticas que está muy próxima a la notación matemática, con lo cual los programas FORTRAN son mucho más sencillos de desarrollar, entender y depurar que los escritos en código máquina o ensamblador. Por otra parte, los esfuerzos empleados en la optimización del compilador dieron su fruto. Se estima que en 1958 aproximadamente la mitad del código escrito para el ordenador IBM 704 estaba escrito en FORTRAN, lo cual muestra la gran aceptación que tuvo el lenguaje.

FORTRAN II apareció en 1958. En el compilador de FORTRAN II se resolvieron algunos errores del compilador de FORTRAN I y se proporcionaron nuevas capacidades, como por ejemplo la posibilidad de compilar separadamente las subrutinas. Una subrutina es un fragmento de código del programa que realiza una tarea específica y relativamente independiente del resto del código. Esto redujo considerablemente el tiempo de compilación, ya que en FORTRAN I las subrutinas debían ser compiladas junto con el programa, con lo cual cada vez que se hacía un cambio debía recompilarse todo el código.

A principios de los años 1960, apareció FORTRAN IV. Añadió nuevas capacidades a las de FORTRAN II y fue uno de los lenguajes de programación más usado de su tiempo. En aquella época, las ventajas del uso de los lenguajes de alto nivel eran ya incuestionables.

Una ventaja de los lenguajes de alto nivel, como FORTRAN, sobre los de bajo nivel (ensamblador y código máquina), es la portabilidad. Un mismo programa escrito en un lenguaje de alto nivel puede ser compilado al código máquina de cualquier máquina, siempre y cuando se disponga de un compilador para el lenguaje que sea adecuado para esa máquina.

A medida que los programas se hicieron más complejos, la eficiencia del código se analizó desde una óptica diferente a simplemente contar el número de instrucciones máquina generadas por el compilador. La facilidad que proporcionan los lenguajes de programación de alto nivel para realizar programas fácilmente comprensibles por el programador (en comparación con los programas en código máquina o ensamblador) influye de manera determinante en la eficiencia del código ejecutable generado. Esto es debido a que la eficiencia de un programa depende de las decisiones que se tomen en todas las fases de su diseño: desde la elección de las estructuras de datos y algoritmos, hasta la codificación final. El lenguaje de programación no sólo afecta a las decisiones que se toman en las diferentes fases del diseño del programa, sino que también afecta a la facilidad con que estas decisiones pueden ser modificadas.

FORTRAN IV fue el estándar de FORTRAN hasta el año 1978, en que apareció FORTRAN 77. FORTRAN es todavía usado ampliamente en aplicaciones de cálculo numérico. El lenguaje ha seguido evolucionando en posteriores versiones (Fortran 90, 95, 2003, 2008 y 2018), añadiendo nuevas funcionalidades como los punteros, el soporte a la orientación a objetos y al código concurrente, etc.

Con el fin de ilustrar qué apariencia tiene un programa escrito en Fortran 90, a continuación se muestra un programa sencillo.

### *Ejemplo*

En este ejemplo se muestra un programa que pide al usuario que introduzca por teclado dos números. A continuación, el programa calcula la suma de estos números, haciendo una llamada a una función definida a tal fin. Finalmente, el programa muestra por pantalla el resultado de la suma. El programa se realiza de dos maneras diferentes:

1. Escribiendo la función y el programa principal en un mismo fichero.
2. Definiendo la función en un fichero y el programa principal en otro. Con ello, se pretende ilustrar cómo se realiza la compilación separada de la función.

La función y el programa principal, definidos ambos en el mismo fichero, se muestran en el Código 1.2. En Fortran 90, no se distingue entre mayúsculas y minúsculas, y los comentarios comienzan con el símbolo `!`. La estructura de los programas es bastante rígida. Por ejemplo, todas las declaraciones de constantes, parámetros y variables van agrupadas al comienzo del programa.

```
!Programa en Fortran 90 almacenado
!en un unico fichero
program CalculoSuma
  implicit none
  real:: x, y, resultado, suma
  write(*,*) 'Introduce el primer sumando'
  read(*,*)x
  write(*,*) 'Introduce el segundo sumando'
  read(*,*)y
  resultado = suma(x,y)
  write(*,*) 'El resultado es: ', resultado
end program CalculoSuma
!Fin del programa principal
function suma(x,y)
  implicit none
  real:: suma, x, y
  suma = x+y
  return
end function suma
```

**Código 1.2:** Programa en Fortran 90 almacenado en un único fichero.

```
!Funcion suma en Fortran 90
function suma(x,y)
  implicit none
  real:: suma, x, y
  suma = x + y
end function suma
```

**Código 1.3:** Función en Fortran 90.

```
!Programa principal en Fortran 90
program CalculoSuma
   implicit none
   real:: x, y, resultado
   real, external::suma
   write(*,*) 'Introduce el primer sumando'
   read(*,*)x
   write(*,*) 'Introduce el segundo sumando'
   read(*,*)y
   resultado = suma(x,y)
   write(*,*) 'El resultado es: ', resultado
end program CalculoSuma
```

**Código 1.4:** Programa principal en Fortran 90.

El programa principal comienza con la palabra reservada **program**, seguida del nombre del programa. El nombre del programa no tiene forzosamente que coincidir con el nombre del fichero donde se almacena dicho programa. El programa principal termina con las palabras reservadas **end program**, seguidas del nombre del programa.

La sentencia **implicit none**, que está justo al inicio del programa principal, indica al compilador que no use las reglas implícitas para tipos y nombres de variables, lo que obliga al programador a declarar todos los nombres de variables.

A continuación de la sentencia **implicit none**, se escriben las sentencias de declaración de parámetros y variables, y tras ellas la parte ejecutable del programa.

En este caso, la parte ejecutable del programa está formada por sentencias de escritura en pantalla, de lectura de los datos proporcionados por el usuario a través del teclado y de una llamada a la función suma, que ha sido declarada al inicio del programa principal y definida a continuación de éste.

Tras el programa principal, se encuentra la definición de la función suma. La definición de la función contiene la declaración de sus argumentos, del tipo de éstos y del valor devuelto por la función. Esto puede ser definido en una sentencia de declaración dentro de la función o, alternativamente, en la primera línea de definición de la función. La sentencia **return** indica el punto en el cual termina la ejecución de la función.

Como se ha explicado anteriormente, Fortran 90 permite la compilación separada. Es decir, que las unidades de programa puedan separarse en diferentes ficheros y ser compiladas por separado. En Código 1.3 y 1.4 se muestra la organización del programa en dos ficheros separados: uno que contiene la definición de la función y el otro el programa principal.

En el programa principal se ha de indicar que existe una función externa, lo que se hace mediante la sentencia:

```
real, external::suma
```

donde `suma` es el nombre de la función, la cual devuelve un valor del tipo **real**.

### 1.2.4. ALGOL

El compilador de FORTRAN I vio la luz en el año 1957. Sin embargo, FORTRAN no era el único lenguaje de alto nivel que se encontraba en desarrollo en

aquella época. Muy al contrario, aparecieron un número considerable de lenguajes de programación diferentes, cuyos compiladores se realizaban únicamente para un cierto tipo de ordenador en concreto: ordenadores UNIVAC, ordenadores IBM de la serie 700, etc.

Pronto se hizo patente que la creciente proliferación de lenguajes de programación suponía un obstáculo para la reutilización y la portabilidad de los programas. Por ello, en el año 1958 se inició un esfuerzo internacional por diseñar un lenguaje estándar universal. El grupo de diseño del nuevo lenguaje estaba integrado por estadounidenses y europeos.

Uno de los requisitos que debía cumplir el nuevo lenguaje era que su sintaxis debería ser tan próxima como fuera posible a la notación matemática, de manera que los programas escritos en este lenguaje fueran fácilmente comprensibles por el programador. Por supuesto, otro objetivo igualmente importante fue que el lenguaje fuera implementable. Es decir, que los programas descritos empleando el nuevo lenguaje pudieran ser traducidos de manera automática a código máquina.

Fruto de este esfuerzo, a finales del año 1958 se dio a conocer la propuesta de un nuevo lenguaje, al que se llamó ALGOL 58. El nombre ALGOL proviene de ALGOrithmic Language.

ALGOL 58 generalizó muchas de sus características de FORTRAN I, y añadió nuevas construcciones y conceptos. Algunas de las nuevas capacidades se desarrollaron con el fin de no ligar el lenguaje de programación a una máquina específica. Así, ALGOL fue el primer lenguaje que se diseñó con la idea de ser independiente de la máquina. Otras nuevas capacidades fueron intentos de hacer el lenguaje más flexible y potente. Algunas de las características de FORTRAN que fueron modificadas en ALGOL son las siguientes:

- Se permitió que los identificadores tuviesen cualquier longitud. En la versión coetánea de FORTRAN estaban restringidos a un número máximo de 6 caracteres.
- Se permitió que las variables vectoriales pudiesen tener cualquier número de dimensiones. En FORTRAN los vectores estaban limitados a un máximo de tres dimensiones.
- El límite inferior de los vectores podía ser especificado por el programador, mientras que en FORTRAN era implícitamente uno.
- Se permitieron las sentencias de selección anidadas, que no estaban permitidas en FORTRAN.

Las modificaciones a ALGOL 58, propuestas durante el año 1959, dieron lugar a que en el año 1960 apareciera una nueva versión del lenguaje: ALGOL 60. Entre estas modificaciones, que fueron de bastante calado, cabe destacar las siguientes:

- Se introdujo el concepto de *bloque de código*, dentro del cual pueden declararse *variables locales* a dicho bloque. Las variables locales a un bloque sólo existen dentro de él, no estando declaradas fuera del bloque. Estos conceptos se describirán en §3.2.2/106.
- Se permitieron las dos formas siguientes de pasar los parámetros a los subprogramas (se explicarán ampliamente en §9.2/364):
  - *Paso por valor*. En el paso del parámetro por valor, se hace una copia en memoria del valor del parámetro y el subprograma trabaja con la copia. Si dicha copia es modificada en el subprograma, el valor original del parámetro no se ve modificado.
  - *Paso por referencia*. Al pasar el parámetro por referencia no se hace una copia de él. El subprograma trabaja directamente con el parámetro actual, pudiendo modificar su valor.
- Se permitió que los procedimientos fuesen *recursivos*. Se dice que un procedimiento es recursivo cuando la llamada al procedimiento puede producir a su vez una o varias llamadas a ese mismo procedimiento. El lenguaje LISP (§1.2.5/42) ya permitía definir funciones recursivas en el año 1959, pero este concepto no era soportado aún por los lenguajes de programación imperativos. La definición de funciones recursivas en los lenguajes imperativos se explicará en §9.3/371.
- Se introdujo una novedad en la declaración del tamaño de los arrays. Un array es un grupo de variables del mismo tipo, al que se hace referencia por medio de un nombre común. El tamaño del array se fija en el momento de su declaración, no pudiendo ser modificado posteriormente. La novedad consistió en que el tamaño del array pudiera especificarse mediante una variable, en lugar de mediante una constante. Así, el número de elementos del array es igual al valor que tenga una variable en el momento de la declaración del array. Se volverá sobre este tema en §3.4/113.

Todos los lenguajes de programación imperativos diseñados desde 1960 pueden considerarse descendientes directos o indirectos de ALGOL 60, ya que incorporan ideas que fueron propuestas por vez primera en ALGOL 60. Entre estos lenguajes,

cabe destacar los diseñados por Niklaus Wirth entre los años 1966 y 1988: ALGOL W, Pascal, Modula-2 y Oberon. También son descendientes de Algol 60, en mayor o menor medida, Ada y Java, así como los lenguajes de la familia del lenguaje C: CPL, Basic CPL, C y C++.

Sin embargo, ALGOL 60 nunca llegó a ser un lenguaje ampliamente usado en Estados Unidos y Europa. Una de las razones es que el lenguaje era muy flexible, con lo cual sus implementaciones eran ineficientes. Otra razón es que ALGOL 60 no proporcionaba sentencias de entrada/salida de datos, sino que éstas dependían de la implementación. Esto suponía un obstáculo para la portabilidad de los programas entre máquinas. A consecuencia de ello, el lenguaje ALGOL jamas llegó a gozar de la popularidad que tuvo entre los programadores el lenguaje FORTRAN.

### 1.2.5. LISP

A mediados de los años 1950 surgió interés por usar los lenguajes de programación en aplicaciones del área de la Inteligencia Artificial (IA). Esto motivó el desarrollo de un nuevo paradigma de programación: la *programación funcional*.

El interés por la IA se inició en diferentes disciplinas, incluyendo la lingüística, la psicología y la matemática. Los lingüistas querían conocer el procesamiento del lenguaje natural. Los psicólogos estaban interesados en modelar cómo el ser humano almacena y recupera la información, así como en otros procesos fundamentales del cerebro. Los matemáticos querían mecanizar ciertos procesos inteligentes, tales como la demostración de teoremas. Todas estas investigaciones llegaron a la misma conclusión: se debía desarrollar un método que permitiese a los ordenadores procesar datos simbólicos en *listas*.

Una *lista* es una secuencia de cero o más valores. Los elementos de la lista pueden ser de varios tipos: Booleanos, números, arrays, símbolos, cadenas de caracteres, funciones, otras listas, etc. Las operaciones básicas sobre una lista son las necesarias para la construcción, inspección y manipulación de la lista. Por ejemplo, buscar elementos que satisfagan una condición, extraer, transformar y eliminar elementos, aplicar una función a elementos de la lista, y obtener ciertas propiedades de la lista, tales como su número de elementos. Se volverá sobre ello en §11.2/459.

El concepto de procesado de listas fue desarrollado por Allen Newell, J. C. Shaw y Herbert Simon, y publicado por primera vez en 1956. Desarrollaron un lenguaje llamado IPL, que continuó evolucionando hasta 1960, pero que no fue muy usado.

En 1958, dos investigadores del Massachusetts Institute of Technology (MIT), llamados John McCarthy y Marvin Minsky, comenzaron a desarrollar el lenguaje LISP. Este lenguaje fue, durante más de 25 años, el más ampliamente usado para el desarrollo de aplicaciones relacionadas con la IA. Actualmente se emplean mucho dos de sus dialectos: COMMON LISP y Scheme.

### 1.2.6. COBOL

COBOL surgió con el propósito de ser un lenguaje estándar para las aplicaciones de negocios, en las cuales se precisa realizar el procesamiento automático de datos. Uno de los principales objetivos del desarrollo fue que el nuevo lenguaje fuese fácil de usar, para lo cual debería ser tan próximo como fuera posible al idioma Inglés.

El desarrollo de COBOL fue impulsado por el Departamento de Defensa de los EE.UU. La primera reunión del grupo de desarrollo del lenguaje tuvo lugar en mayo de 1959, en el Pentágono. COBOL fue estandarizado por el American National Standards Institute (ANSI) en 1968. Las dos siguientes versiones fueron estandarizadas por ANSI en 1974 y 1985.

El lenguaje COBOL ha sido muy ampliamente usado y aún lo es en la actualidad. Asimismo, algunos de los conceptos que incorporó fueron adoptados por otros lenguajes desarrollados posteriormente. Por ejemplo, la sentencia **DEFINE** de COBOL 60 fue la primer construcción para macros de un lenguaje de alto nivel. También, COBOL fue el primer lenguaje en soportar estructuras de datos jerárquicas.

### 1.2.7. Prolog

El concepto de la *programación lógica* ha estado históricamente ligado a un lenguaje denominado Prolog (PROgramming LOGic). El lenguaje Prolog fue desarrollado, a comienzos de los años 1970, por un grupo de investigación compuesto por miembros de los Departamentos de Inteligencia Artificial de las Universidades de Marsella y Edimburgo. El primer intérprete de Prolog apareció en 1972.

El concepto que subyace tras la programación lógica consiste en emplear *hechos* y *reglas* para representar la información, y usar deducción para responder preguntas.

Las *reglas* son relaciones del tipo:

```
P if Q1 and Q2 and ... and Qn.
```

que significa que el término `P` es verdadero si se satisfacen `Q1`, ..., `Qn`.

Los *hechos* son un tipo especial de regla, en la cual se verifica `P` sin necesidad de que se satisfaga ninguna condición. Los hechos se escriben simplemente:

```
P.
```

El lenguaje de programación lógico facilita la descripción de estas reglas y también de las preguntas a las que se desea responder aplicando dichas reglas. Como resultado de la ejecución del programa, se obtiene la respuesta a dichas preguntas mediante deducción, a partir de las reglas y los hechos.

### 1.2.8. BASIC y Visual BASIC

BASIC (Beginners' All-purpose Symbolic Instruction Code) es una familia de lenguajes de programación cuya filosofía de diseño es la facilidad de uso. La versión original fue diseñada por John G. Kemeny y Thomas E. Kurtz y lanzada en Dartmouth College en 1964.

Kemeny y Kurtz desarrollaron, además de BASIC, el Dartmouth Time Sharing System, que permitió a múltiples usuarios editar y ejecutar simultáneamente programas en BASIC. BASIC fue muy popular en los microcomputadores de finales de los 70 y principios de los 80. Tenía la ventaja de que era fácil de aprender y que sus dialectos se podían implementar en ordenadores con pequeñas memorias. Su uso decayó cuando aumentaron las capacidades de los microcomputadores y surgieron nuevos lenguajes de programación.

El uso de BASIC resurgió con la aparición de Visual Basic (VB) al inicio de los años 90. VB fue muy usado porque proporcionó una manera sencilla de construir interfaces gráficas de usuario. Cuando .NET apareció, surgió con él una nueva versión de VB llamada VB.NET. La versión .NET soporta completamente la programación orientada a objetos.

### 1.2.9. SIMULA 67

SIMULA 67, desarrollado por los noruegos Kristen Nygaard y Olen Johan Dahl, fue presentado públicamente en 1967. Tenía como objetivo facilitar la simulación por ordenador de modelos de eventos discretos.

Este lenguaje, que es una extensión del ALGOL 60, fue el primero en presentar conceptos de orientación a objetos, ya que introdujo los conceptos de *clase*, *objeto* (instanciación de una clase) y *herencia*. Una clase en SIMULA 67 consiste en una

colección de procedimientos y declaraciones de variables. Cada vez que se crea un objeto de una clase, se asigna memoria para contener la correspondiente colección de dichas variables. Algunos conceptos fundamentales de la programación orientada a objetos se explicarán en §1.3.4/59.

SIMULA también introdujo el concepto de *puntero* y la *gestión dinámica de la memoria*. Un puntero es una variable que almacena la dirección de memoria en la cual está almacenado un objeto, de manera que puede emplearse el puntero para referenciar al objeto apuntado. Los punteros se explicarán en §3.6/117 y la gestión dinámica de la memoria en §3.7/118.

### 1.2.10. Pascal

Los lenguajes de programación usados ampliamente en la década de 1960, como era el caso de FORTRAN, proporcionaban un conjunto muy limitado de sentencias para el control del flujo del programa, estando éste basado en el uso de sentencias de salto (sentencias *goto*).

En la década de 1970 se popularizó un estilo de programación denominado *programación estructurada*. Uno de los objetivos que se perseguían era mejorar la legibilidad de los programas, en gran medida dificultada por el uso indiscriminado en los programas de sentencias de salto. Téngase en cuenta que una legibilidad pobre de los programas hace difícil depurar y mantener el código. Los programas basados en el empleo de sentencias de salto se denominaron peyorativamente *código espagueti*: el flujo del programa se asemejaba en lo enmarañado a un plato de espagueti.

Pascal, basado en ALGOL 60, fue desarrollado por Niklaus Wirth y presentado públicamente en 1971. El lenguaje Pascal facilitaba la *programación estructurada*, cuyo fundamento reside en la observación siguiente: un programa que puede ser leído de arriba abajo es más sencillo de entender que un programa que va saltando de unas partes a otras.

Debido a las facilidades que Pascal proporciona para la práctica de la programación estructurada, Pascal fue usado ampliamente, desde mediados de 1970 hasta finales de 1990, para enseñar programación en las universidades.

Se desarrollaron varios dialectos de Pascal, que pretendían extender las capacidades del lenguaje. Uno de ellos fue Turbo Pascal, de la compañía Borland. Posteriormente, esta misma compañía creo el lenguaje Delphi, añadiendo a Pascal soporte para la orientación a objetos y facilidades para la programación de interfaces gráficas de usuario.

```
{Programa en Pascal almacenado
en un unico fichero}
Program CalculoSuma;
var
  x, y, result: real;
  Function suma (x: real; y:real):real;
  begin
    suma := x + y;
  end;
begin
  writeln('Introduce el primer sumando');
  readln(x);
  writeln('Introduce el segundo sumando');
  readln(y);
  result := suma(x,y);
  writeln('El resultado es: ', result);
end.
```

**Código 1.5:** Programa en Pascal que calcula la suma de dos números.

Para ilustrar la forma que tiene un programa escrito en lenguaje Pascal, a continuación se muestra un ejemplo. Este programa tiene la misma funcionalidad que el escrito anteriormente en Fortran 90.

### *Ejemplo*

El programa principal y la función, almacenados en un mismo fichero, se muestran en Código 1.5. En Pascal, al igual que en Fortran 90, no se distingue entre mayúsculas y minúsculas, y los programas tienen una estructura rígida. Es preciso declarar las variables antes de usarlas y todas las declaraciones han de ir precedidas de la palabra reservada **var**.

La primeras dos líneas del programa son un comentario. En Pascal, los comentarios se escriben encerrados entre llaves. El programa debe empezar con la palabra reservada **program**, seguida del nombre del programa, y finalizar con la palabra reservada **end.** (obsérvese que finaliza en punto). El cuerpo del programa principal se escribe delimitado por las palabras reservadas **begin** y **end.**.

En Pascal, las funciones han de declararse fuera del cuerpo del programa principal. El cuerpo de la función se delimita mediante las palabras reservadas **begin** y **end.**. En la primera línea de definición de la función, se declara el nombre y el tipo de los argumentos de la función, así como el tipo del valor que devuelve la función. En la función pueden declararse *variables locales*, que sólo existen dentro del cuerpo de

la función y no son accesibles desde el programa que llama a la función. La función `suma` no tiene ninguna variable local.

En el cuerpo del programa principal se encuentran las sentencias de ejecución del programa, que finalizan siempre en punto y coma. Las asignaciones emplean el símbolo `:=`, en lugar del símbolo `=` empleado en FORTRAN. Dentro del cuerpo del programa principal se realiza una llamada a la función `suma`.

Pascal también permite la compilación separada, por lo que el programa podría escribirse en tres ficheros separados: uno en el cual se declarase la función, otro que contuviera la definición de la función y finalmente otro con el programa principal. Mediante una directiva del lenguaje, sería necesario incluir el fichero con la declaración de la función en los otros dos ficheros.

### 1.2.11. C

El lenguaje C fue diseñado e implementado por Dennis Ritchie, en los laboratorios Bell, en 1972. En 1989, ANSI publicó una descripción estándar de C. En las décadas de 1980 y 1990, el lenguaje C alcanzó una gran popularidad, debido en gran medida al hecho de que el sistema operativo UNIX proporciona un compilador de C. A finales de la década de 1980, se desarrolló una nueva versión de C, denominada C++ (§1.2.15/49), que facilita la programación orientada a objetos.

### 1.2.12. Modula-2

El lenguaje Modula fue diseñado por Niklaus Wirth en 1976. Sin embargo, no llegó a desarrollarse ningún compilador para ese lenguaje. Wirth continuó desarrollando el lenguaje, diseñando a mediados de los años 1980 un nuevo lenguaje, denominado Modula-2, que estaba basado en Pascal y en Modula. Modula-2 nunca ha llegado a ser un lenguaje ampliamente usado por los programadores. Sin embargo, fue muy ampliamente usado en la enseñanza de la programación.

### 1.2.13. Ada

Ada es el resultado del más largo y caro esfuerzo de diseño de un lenguaje de programación jamás llevado a cabo. El desarrollo del lenguaje, que fue encargado por el Departamento de Defensa de EE.UU., comenzó a mediados de los años 1970

y terminó en 1983. Cuatro de las principales características del lenguaje Ada son las siguientes:

- Permite declarar paquetes (*packages*), dentro de los cuales pueden declararse objetos, tipos de datos y procedimientos.
- Proporciona facilidades para el tratamiento de las *excepciones*. Es decir, de los errores en tiempo de ejecución. El lenguaje permite al programador describir qué acciones deben realizarse si se produce determinado tipo de excepción durante la ejecución de un fragmento de código.
- Las unidades de programa (subprogramas o paquetes) pueden ser genéricas. Es decir, existen características de los parámetros que se puede precisar al instanciar la unidad. Las unidades genéricas no se usan directamente, sino que se tienen que instanciar previamente. Estas representan una plantilla mediante la cual se indica al compilador cómo poder crear unidades no genéricas que tienen el mismo funcionamiento que la unidad genérica. Las unidades genéricas son análogas a las plantillas (*templates*) existentes en C++.
- Permite la ejecución concurrente de unidades de programa especiales, llamadas tareas (*tasks*).

Ada continuó evolucionando, definiéndose los estándares Ada 95 en 1995 y posteriormente Ada 2005. Su popularidad disminuyó principalmente por dos motivos. El Departamento de Defensa dejó de requerir su uso en sistemas software militares. Además, el lenguaje C++ empezó a tener una aceptación muy amplia para la programación orientada a objetos incluso antes de que se lanzara Ada 95.

### 1.2.14. Smalltalk

Smalltalk se desarrolló en los años 1970, en el centro de investigación de Xerox en Palo Alto (EE.UU.), por un grupo liderado por Alan Kay. Aparecieron diferentes versiones del lenguaje, siendo Smalltalk 80 la versión definitiva.

Si bien los conceptos de clase, objeto y herencia fueron introducidos en SIMULA 67, Smalltalk es considerado el primer lenguaje que facilita la aplicación del paradigma de la programación orientada a objetos. Asimismo, con Smalltalk surgieron por vez primera los sistemas de ventanas, que hoy en día son comunes en las interfaces de usuario.

### 1.2.15. C++

C++ fue desarrollado por Bjarne Stroustrup, en los laboratorios Bell, en 1980. Es un lenguaje que ha evolucionado a partir de C, a través de una serie de modificaciones para mejorar sus características imperativas y de adiciones para el soporte de la programación orientada a objetos. C++ es casi completamente compatible con C. Es decir, la inmensa mayoría de los programas escritos en C pueden ser compilados como programas en C++. A continuación, se muestra el programa en C++ que calcula la suma de dos números.

C++ está estandarizado por la International Organization for Standardization (ISO). En 2011, C++11 inició un ciclo trienal de lanzamiento de nuevas versiones: a C++11 le siguió C++14 y luego C++17.

Microsoft lanzó en el año 2002 su plataforma de computación .NET, que incluyó una nueva versión de C++ llamada Managed C++ o MC++. MC++ extendió C++ para proporcionar acceso a la funcionalidad de la plataforma .NET. También lanzó el lenguaje C#, basado en C++ y Java, e incluyendo ideas de Delphi y Visual Basic.

#### *Ejemplo*

El programa escrito en C++ se muestra en Código 1.6. Al contrario que Fortran 90 y Pascal, C++ sí distingue entre los caracteres en mayúsculas y en minúsculas. Los programas escritos en C++ tienen una estructura más flexible. Por ejemplo, las variables pueden declararse en cualquier punto del programa, siempre y cuando sean declaradas antes de ser usadas. Al igual que en Pascal, las sentencias finalizan con punto y coma. Los comentarios en el código pueden escribirse de dos maneras:

1. Pueden escribirse una o varias líneas de comentarios, escribiendo el símbolo `/*` antes de la primera línea de comentario y el símbolo `*/` después de la última.
2. Cuando el comentario es una única línea o la parte final de una línea, puede señalarse mediante el símbolo `//`. Es un comentario todo lo que se escribe en una línea desde el símbolo `//` hasta el final de la línea.

La primera línea de código tras los comentarios es la directiva:

```
#include <iostream>
```

que indica al compilador dónde encontrar ciertas funciones, de la librería estándar de C++, que se usan en el programa. En este caso, `iostream` es el nombre de una

```
1  /*
2     Programa sencillo en C++ escrito en un solo fichero
3   */
4
5  //Cabeceras
6  #include <iostream>
7  using namespace std;
8
9  // Prototipo de la función
10 double suma(double, double);
11
12 // Variables globales
13 double x, y;
14
15 // Programa principal: función main
16 int main() {
17   double result;
18   cout << "Introduce el primer sumando";
19   cin >> x;
20   cout << "Introduce el segundo sumando";
21   cin >> y;
22   result = suma(x, y);
23   cout << "El resultado es: \n";
24   cout << result;
25   return 0;
26 }
27
28 // Definición de la funcion suma
29 double suma(double x1, double y1) {
30   double result;   //variable local
31   result = x1 + y1;
32   return result;
33 }
```

**Código 1.6:** Programa en C++ almacenado en un único fichero.

cabecera estándar, que contiene las definiciones de las funciones que facilitan la entrada desde el teclado y la salida a la consola. Como se explicará en §1.4.2/63, el programa enlazador (*linker*) combina el código objeto de la librería y el del programa en C++. Las directivas comienzan con el símbolo almohadilla. Algunos compiladores requieren que no existan espacios antes ni después de este símbolo.

La sentencia:

```
using namespace std;
```

permite usar las entidades (tipos de datos, variables, funciones, etc.) del espacio de nombres `std` sin necesidad de especificar que pertenecen a dicho espacio de nombres. En el espacio de nombres `std` están agrupadas las declaraciones de la librería estándar.

Si en el programa no se incluye la sentencia anterior, al hacer referencia a cualquier entidad de la librería estándar habría que anteponer `std::` al nombre de la entidad. Por ejemplo, debería escribirse `std::cin` en lugar de `cin`. En general, en el código C++ de esta Unidad Didáctica no se usará `using namespace std;`. En su lugar, en aquellos casos en que se usen entidades declaradas en la librería estándar, se indicará explícitamente, anteponiendo `std::` al nombre de la entidad.

Los *espacios de nombres* permiten agrupar declaraciones de entidades (tipos de datos, variables, funciones, etc.) bajo un mismo nombre. De esta manera, el ámbito global se divide en subámbitos, cada uno de los cuales tiene su propio nombre. El propósito de los espacios de nombres es evitar que se produzcan "colisiones de nombres". Es decir, que en diferentes partes del programa, dentro del mismo ámbito, se declaren varias entidades con el mismo nombre. La definición de los espacios de nombres en C++ se explicará en §10.11/418.

El programa principal es una función llamada `main`. Mediante:

```
int main()
```

se indica que el programa principal empieza en ese punto. La definición del programa principal está delimitada por llaves. El programa principal usa las variables `x` e `y`, que han sido declaradas antes del inicio del programa y son variables globales. También se declara la función suma antes del inicio del programa, aunque su definición se encuentra tras el programa principal.

Dentro del programa principal hay sentencias para la salida y entrada de datos, y una llamada a la función suma. La línea `return 0` indica el final del programa. Esta línea no tiene forzosamente que ser la última línea del programa principal, aunque es su sitio habitual en programas sencillos. Algunos compiladores permiten omitir esta línea, ya que asumen que el programa ha llegado a su fin cuando no hay más sentencias que ejecutar.

En el Código 1.7, 1.8 y 1.9 se muestra el programa organizado en tres ficheros separados, que contienen respectivamente la declaración de la función (fichero `externsC.h`), la definición de la función (fichero `suma.cpp`) y el programa principal (fichero `CalculoSuma.cpp`). El fichero con la declaración de la función se incluye en los otros dos ficheros empleando la directiva:

```
#include "externsC.h"
```

La organización en varios ficheros de los programas escritos en C++ se explicará en §10.10/416.

```
/*
 * Fichero externsC.h
 */
#ifndef EXTERNSC_H_
#define EXTERNSC_H_

//Prototipo de la función
double suma(double, double);

#endif /* EXTERNSC_H_ */
```

**Código 1.7:** Fichero con el prototipo de la función.

```
/*
 * Fichero suma.cpp
 */
#include "externsC.h"

//Definición de la funcion suma
double suma(double x1, double y1) {
  double result;  //variable local
  result = x1 + y1;
  return result;
}
```

**Código 1.8:** Fichero con la definición de la función.

```
/*
 * Fichero CalculoSuma.cpp
 * Programa principal
 */

//Cabeceras
#include <iostream>
#include "externsC.h"
using namespace std;

// Variables globales
double x, y;

// Programa principal: función main
int main() {
  double result;
  cout << "Introduce el primer sumando";
  cin >> x;
  cout << "Introduce el segundo sumando";
  cin >> y;
  result = suma(x, y);
  cout << "El resultado es: \n";
  cout << result;
  return 0;
}
```

**Código 1.9:** Fichero con el programa principal.

### 1.2.16. Java

Java fue desarrollado en Sun Microsystems en los años 1990, por un equipo liderado por James Gosling. Inicialmente fue diseñado para la programación de dispositivos electrónicos empotrados, para los cuales la fiabilidad es un requisito importante. Cuando el World Wide Web (WWW) empezó a ser ampliamente usado, a partir de 1993, se encontró que Java era útil para la programación web.

Java está inspirado en C++, pero diseñado específicamente para ser más pequeño, simple y fiable. Sus diseñadores eliminaron muchas de las características que hacen inseguro a C++, como el no comprobar en los accesos a un array que el valor del índice está dentro de rango. Java es un lenguaje portable. Los programas escritos en Java pueden ejecutarse en cualquier máquina que tenga instalada una máquina virtual Java. La Máquina Virtual de Java se describe en §1.4.3/65 (sistema híbrido).

### 1.2.17. Lenguajes de scripting

El lenguaje sh, cuyo nombre proviene de la palabra inglesa Shell, es el primer lenguaje de scripting que surgió. Los lenguajes de scripting son interpretados, es decir, para ejecutar las instrucciones existe un programa o intérprete que se encarga de procesar cada una de las órdenes y producir los resultados deseados. La interpretación pura se explicará en §1.4.1/62.

El lenguaje sh comenzó como una pequeña colección de comandos que eran interpretados como llamadas a un sistema de subprogramas que realizaban funciones como gestión y filtrado de ficheros. Este lenguaje tenía variables, sentencias de control de flujo, funciones y otras capacidades que hacía de éste un lenguaje de programación completo.

Otro lenguaje de scripting es awk, desarrollado por Kernighan y Weinberger en los Laboratorios Bell. Este lenguaje comenzó como un lenguaje para generación de informes, pero evolucionó hacia un lenguaje de propósito general. Asimismo, uno de los lenguajes de scripting más potente y conocido es ksh, desarrollado por David Korn en los Laboratorios Bell.

El lenguaje Perl, desarrollado por Larry Wall, fue originariamente una combinación de sh y awk. Se usó inicialmente como una utilidad del sistema operativo UNIX para el procesado de ficheros de texto, pero se usa ahora como un lenguaje de propósito general para una variedad de aplicaciones, tales como la biología computacional y la inteligencia artificial.

### 1.2.18. JavaScript

JavaScript fue desarrollado originariamente por Brendan Eich, en Netscape. A finales de los años 90, la European Compute Manufacturers Association (ECMA) desarrolló un lenguaje estándar para JavaScript: ECMA-262. Este estándar fue también aprobado por la International Standards Organization (ISO) como ISO-16262. La versión de Microsoft de JavaScript se llama JScript.NET.

Habitualmente el código JavaScript se embebe en los documentos HTML y es interpretado por el navegador Web cuando se muestra dicho documento. El uso principal de JavaScript en la programación Web es validar la entrada de datos y crear documentos HTML dinámicos.

### 1.2.19. PHP

PHP fue desarrollado por Rasmus Lesdorf, un miembro del grupo Apache, en 1994. PHP es un lenguaje de scripting específicamente desarrollado para aplicaciones Web. El código PHP se interpreta en el servidor Web cuando un documento HTML en el que está embebido es solicitado por un navegador. El código PHP normalmente produce como salida código HTML, que reemplaza al código PHP del documento HTML. Por tanto, un navegador web nunca llega a ver código PHP.

### 1.2.20. Python

Python es un lenguaje de scripting con orientación a objetos, cuya primera versión data de 1991. Fue desarrollado por Guido van Rossum, del Instituto Nacional de Investigación de Matemáticas y Ciencias de la Computación en Ámsterdam.

Python es un lenguaje de programación muy popular, especialmente para computación científica, ciencia de datos e inteligencia artificial. Es un lenguaje de código abierto, que cuenta con una amplia librería estándar, miles de librerías de código abierto y muchas aplicaciones de código abierto. Asimismo, soporta algunos de los paradigmas de programación más populares, facilita la programación concurrente y se usa para el desarrollo web. Algunas aplicaciones desarrolladas en Python son Dropbox, YouTube y Reddit.

La implementación más usada de Python, Cpython (escrita en lenguaje C), usa una mezcla de compilación e interpretación para ejecutar los programas.

***Ejemplo***

El Código 1.10 es un programa en Python que calcula la suma de dos números.

```
1 def suma(x1, y1):
2       return x1+y1
3 x = int(input('Introduce el primer sumando: '))
4 y = int(input('Introduce el segundo sumando: '))
5 result = suma(x, y);
6 print('El resultado es: ', result)
```

**Código 1.10:** Programa en Python que suma dos números.

La primera línea de código constituye la declaración de la función `suma`. La declaración de una función en Python comienza con la palabra reservada **def**, seguida del nombre de la función, un par de paréntesis y los dos puntos.

En el lenguaje Python se distingue entre minúsculas y mayúsculas. Por convenio, los nombres de la función deberían tener la inicial en minúscula. La lista de parámetros se escribe entre paréntesis, separados por comas. El cuerpo de la función ha de estar indentado.

En Python un bloque de código (cuerpo de una función, bucle, etc.) empieza con una indentación y finaliza con la primera línea no indentada. La indentación puede consistir en un número variable de espacios en blanco, pero debe ser la misma dentro del mismo bloque.

Después de la función, se le asigna valor a las variables x e y. Los nombres que identifican a las variables pueden ser letras, dígitos y guión bajo, pero no pueden comenzar por un dígito.

La función **input**, integrada en Python, se emplea para solicitar y obtener la entrada del usuario. Primero, la función muestra el *string* (cadena de caracteres entre comillas simples) que hemos pasado como argumento a la función. Entonces, la función se queda esperando a que el usuario responda. La salida de la función **input** es siempre una variable de tipo *string*.

En este caso, hemos convertido el *string* a una variable de tipo entero, usando la función predefinida **int**. En la Línea 5 del código, se invoca la función `suma`, pasando como argumentos las variables x e y. El resultado se almacena en la variable `result`. Finalmente, se usa la función predefinida **print** para escribir en la consola el literal `'El resultado es: '` seguido del valor de la variable `result`.