Apuntes de Metodología y Tecnología de la Programación, IS04

Curso 2006/2007

Gloria Martínez Vidal, Miguel Pérez Francisco

27 de septiembre de 2006

Reconocimiento-NoComercial-CompartirIgual 2.0

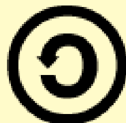
Usted es libre de:

- copiar, distribuir y comunicar públicamente la obra
- hacer obras derivadas

Bajo las condiciones siguientes:

Reconocimiento. Debe reconocer y citar al autor original.

No comercial. No puede utilizar esta obra para fines comerciales.



Compartir bajo la misma licencia. Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- Alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor

Los derechos derivados de usos legítimos u otras limitaciones no se ven afectados por lo anterior.

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/2.0/ or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Esta versión de los apuntes surge del trabajo realizado en la asignatura *“Metodología y Tecnología de la Información, IS04”* de la titulación “Ingeniería Técnica en Informática de Sistemas” de la Escuela Superior de Tecnología y Ciencias Experimentales de la Universitat Jaume I, durante los cursos 2001/02, 2002/03 y 2003/04.

Gran parte del material procede de los apuntes de otra asignatura, *“Introducción a la Algorítmica, E45”*, realizados por el profesor Luis Amable García Fernández y la profesora Gloria Martínez Vidal. Y, por supuesto, en su elaboración los autores han recibido la ayuda y los consejos de sus compañeros, especialmente de María de los Ángeles López Malo, Mar Marcos López, Luis Amable García Fernández, Teresa Escrig Monferrer, Javier Olcina Velamazán e Ismael Sanz Blasco. A ellos, y a todos los demás compañeros que de alguna forma nos ayudaron y alentaron, muchas gracias.

Índice general

1. Introducción **1**

1.1. Conceptos Básicos de Programación. 1

1.1.1. Conceptos Básicos. 2

1.2. La Programación como Disciplina de Ingeniería. 5

1.2.1. Etapas del Desarrollo del Software. 6

1.2.2. Objetivos de la Asignatura. 7

1.3. Resumen. 8

1.4. Glosario. 9

1.5. Bibliografía. 9

2. Conceptos Básicos **11**

2.1. Variables. Tipos Básicos y Expresiones. 11

2.2. C: Estructura Básica de un Programa, Variables y Tipos Básicos. 17

2.2.1. Ejemplo de Programa en C. 17

2.2.2. Tratamiento de Variables y Tipos Básicos. 19

2.2.3. Entrada/Salida de Datos. 24

2.3. Resumen y Consideraciones de Estilo. 28

2.4. Glosario. 28

2.5. Bibliografía. 29

2.6. Actividades y Problemas Propuestos. 29

3. Programación Estructurada **31**

3.1. Introducción a la Programación Estructurada. 32

3.1.1. Evolución de los Computadores. . . . 32

3.1.2. Programación Estructurada. . . . 34

3.1.3. El Concepto de Predicado. Cálculo de Predicados. . . . 35

3.2. Esquemas Condicionales. . . . 36

3.2.1. Ejemplos del Uso de Esquemas Condicionales. . . . 38

3.2.2. Esquemas Condicionales en C. . . . 41

3.3. Esquemas Iterativos. . . . 45

3.3.1. Ejemplos del Uso de Esquemas Iterativos. . . . 46

3.3.2. Potencial y Peligro de los Bucles Condicionales. . . . 48

3.3.3. Bucles con Contador. . . . 50

3.3.4. Esquemas Iterativos en C. . . . 51

3.4. Lógica de las Estructuras de Control. . . . 54

3.4.1. Lógica del Esquema Condicional. . . . 56

3.4.2. Lógica del Esquema Iterativo. . . . 60

3.5. Resumen y Consideraciones de Estilo. . . . 64

3.6. Glosario. . . . 65

3.7. Bibliografía. . . . 66

3.8. Actividades y Problemas Propuestos. . . . 66

4. Introducción al Diseño Descendente 77

4.1. Acciones No Primitivas. Concepto de Parámetro. . . . 78

4.1.1. Acciones No Primitivas. . . . 79

4.1.2. Concepto de Parámetro. . . . 80

4.1.3. Acciones no Primitivas en C. . . . 82

4.2. Especificación de Acciones no Primitivas. . . . 93

4.2.1. Dominio de Definición. . . . 94

4.3. Introducción al Diseño Descendente: Estructuración y Reutilización del Trabajo. . . . 96

4.3.1. Un Ejemplo en C. . . . 97

4.3.2. Definición y Uso de Bibliotecas en C. . . . 110

4.4. Resumen y Consideraciones de Estilo. . . . 111

4.5. Glosario. 112

4.6. Bibliografía. 112

4.7. Problemas Propuestos. 113

5. Estructuras de Datos Estáticas 119

5.1. Necesidad de las Estructuras de Datos. 120

5.1.1. Ampliación de la Noción de Tipo de Datos. 121

5.1.2. Clasificación de las Estructuras de Datos. 122

5.2. Vectores. 122

5.2.1. Necesidad de los Vectores. 123

5.2.2. Declaración de vectores en `C`. 124

5.2.3. Operaciones elementales de acceso en `C`. 126

5.2.4. Cadenas de caracteres. 128

5.2.5. Vectores Multidimensionales. Matrices. 129

5.3. Tuplas 131

5.3.1. Necesidad de las tuplas 132

5.3.2. Declaración de tuplas en `C`. 132

5.3.3. Operaciones elementales de acceso en `C`. 134

5.3.4. Combinación de distintos tipos 136

5.4. Algoritmos de manipulación de las estructuras de datos 137

5.4.1. Esquema de Recorrido 137

5.4.2. Esquema de Búsqueda 141

5.5. Resumen y Consideraciones de Estilo. 142

5.6. Glosario. 143

5.7. Bibliografía. 144

5.8. Problemas Propuestos. 144

6. Estructuras de Datos Dinámicas 157

6.1. Introducción. 157

6.2. Memoria Dinámica. Concepto de puntero 158

6.2.1. Trabajo con punteros en C. 160
6.2.2. El lenguaje C y los punteros. 163
6.3. Estructuras de Datos Dinámicas. 167
6.3.1. Gestión dinámica de la memoria. 168
6.3.2. Estructuras enlazadas. 170
6.3.3. Ejemplo: Definición de una lista simplemente enlazada. 175
6.4. Glosario. 182
6.5. Resumen y Consideraciones de Estilo. 183
6.6. Bibliografía. 184
6.7. Problemas Propuestos. 184

7. Introducción al Uso de Ficheros **191**

7.1. Introducción 192
7.2. Manipulación básica de ficheros en C. 193
7.2.1. La tupla FILE. 193
7.2.2. Apertura y cierre de ficheros. 194
7.3. Ficheros de texto. 196
7.3.1. Escritura en un fichero de texto 196
7.3.2. Lectura en un fichero de texto 198
7.4. Ficheros binarios. 198
7.4.1. Escritura en un fichero binario. 199
7.4.2. Lectura en un fichero binario. 200
7.4.3. Un ejemplo: escritura y lectura de una lista. 201
7.5. Ficheros de acceso directo. 205
7.6. Resumen y Consideraciones de Estilo. 206
7.7. Bibliografía. 206
7.8. Problemas Propuestos. 207

8. Otros Estilos de Programación **211**

8.1. Introducción 211

8.1.1. Breve Perspectiva Histórica de los Lenguajes de Programación 212

8.1.2. Metodología y Tecnología de la Programación 215

8.2. Programación Tradicional y Programación Moderna. 216

8.3. Programación Orientada a Objetos 220

8.3.1. ¿Qué es? 220

8.3.2. Primer Paso: Programación Procedural. 220

8.3.3. Segundo Paso: Programación Modular. 222

8.3.4. Tercer Paso: Programación Orientada a Objetos. 226

8.3.5. Eventos y Excepciones. 229

8.4. Python: Tipos Básicos y Estructuras de Control. 231

8.4.1. Introducción. ¿Qué es Python y por qué Python? 231

8.4.2. Tratamiento de Variables y Tipos Básicos en Python. 232

8.4.3. Estructuras de Control de Flujo de Datos. 236

8.4.4. Acciones no Primitivas en Python. 241

8.5. Python: Gestión de la Información. 247

8.5.1. Tipos Compuestos: Secuencias. 247

8.5.2. Ficheros en Python. 256

8.6. Python: Actualizaciones. 257

8.7. Agradecimientos. 258

8.8. Glosario. 258

8.9. Bibliografía. 259

9. Recursividad 261

9.1. ¿Qué es la Recursividad?. 261

9.1.1. El Concepto de Recursividad en Algorítmica. 262

9.1.2. Ejemplos de Algoritmos Recursivos. 265

9.2. Tipos de Recursividad. 267

9.3. La Recursividad en el Computador. 268

9.3.1. Utilidad de la Recursividad. 269

9.4. Lecturas Recomendadas. 272

9.5. Problemas Propuestos. 273

10. Introducción al Análisis de Algoritmos 281

10.1. Introducción. 281

10.1.1. Determinación de la Complejidad Computacional: Tamaño del Problema y Operación Elemental. 283

10.1.2. Algunos Ejemplos Típicos del Cálculo de la Función de Coste. 285

10.2. Análisis en el Peor y en el Mejor Caso. 286

10.3. Órdenes de Complejidad. 290

10.4. ¿Por Qué se Buscan Algoritmos Eficaces? 294

10.4.1. Clases de Problemas. 295

10.5. Algoritmos de Ordenación. 297

10.5.1. Ordenación por Selección (Selection Sort). 298

10.5.2. Método de la Burbuja (Bubble Sort). 300

10.5.3. Ordenación por Inserción (Insertion Sort). 302

10.5.4. Ordenación Rápida (QuickSort). 305

10.6. Bibliografía. 310

10.7. Problemas Propuestos. 310

Capítulo 1

Introducción

Índice General

1.1. Conceptos Básicos de Programación. **1**
1.1.1. Conceptos Básicos. 2
1.2. La Programación como Disciplina de Ingeniería. **5**
1.2.1. Etapas del Desarrollo del Software. 6
1.2.2. Objetivos de la Asignatura. 7
1.3. Resumen. **8**
1.4. Glosario. **9**
1.5. Bibliografía. **9**

"La verdadera diferencia entre el hardware y el software es que el hardware se vuelve más rápido, pequeño y barato con el tiempo mientras que el software se vuelve más grande, lento y caro."
Origen desconocido. Probablemente apócrifo.

1.1. Conceptos Básicos de Programación.

El desarrollo de *software* es una de las tareas más importantes dentro del ámbito de la informática. En esta asignatura se pretende sentar las bases que permitan introducirse en el mundo de la programación, aprender a construir algoritmos que permitan resolver problemas concretos.

Es frecuente concebir la programación como una labor que consiste en aprender la sintaxis de un lenguaje y, a partir de ahí, adquirir una experiencia, mediante el "entrenamiento", que permita ir generando programas más y más complejos. Esta es una visión errónea, si bien extendida, que conviene erradicar. La programación necesita de una técnica y es una disciplina que se debe intentar sistematizar al máximo.

Aprender a programar no es una tarea sencilla. Entre otras cosas porque por mucho que se hable de sistematizar, el diseño de un algoritmo es, en última instancia, un proceso creativo. Pero, evidentemente, no es lo mismo acometer este proceso creativo de forma improvisada que siguiendo una determinada metodología. Si se establece una analogía, por ejemplo, con la escritura de una novela parece evidente que a un novelista profesional se le suponen una serie de conocimientos específicos

(análisis literario, conocimiento de diversos estilos, estructura de contenidos, dominio de la sintaxis y de la semántica. . . y haber leído y escrito mucho) que le ayudarán a crear un buena obra y no le harán caer en errores propios de un escritor aficionado.

Uno de los objetivos de este tema es, precisamente, introducir el mundo de la programación como una disciplina de la ingeniería, a fin de determinar claramente cuáles son los objetivos de un buen programador y poder establecer, por lo tanto, los objetivos fundamentales de la asignatura. Para comprender bien dicha técnica, se comenzará por explicar cuáles son los conceptos básicos que se manejarán a lo largo de toda la asignatura, ya que conviene saber de qué se habla antes de entrar en materia. A partir de estos conceptos, se puede avanzar y describir cómo se debe entender y cómo se debe emprender el desarrollo de un programa. Una vez situados, ya se estará en condiciones de comenzar a conocer cuáles son las herramientas básicas de un programador.

1.1.1. Conceptos Básicos.

El primero de los conceptos es uno de los más importantes en informática y sobre él recaerá la mayor parte del peso de la asignatura:

Definición 1.1 (Algoritmo) *Descripción no ambigua y precisa de una secuencia de acciones que permite resolver un problema bien definido en tiempo finito.*

Por lo tanto, básicamente un algoritmo es la descripción de cómo resolver el problema; pero, además, en la definición aparecen una serie de calificativos que nos indican qué características se están imponiendo a esta descripción:

No ambigua Cada acción debe tener una única interpretación,

Precisa Cada acción debe estar definida rigurosamente.

Como consecuencia de estas dos premisas se tiene que el *lenguaje natural* no es un lenguaje apropiado para definir algoritmos, ya que puede dar lugar a interpretaciones según el oyente.

Secuencia No se refiere sólo a una sucesión más o menos larga de instrucciones: el orden que ocupa una instrucción en esa secuencia es significativo,

Problema bien definido Debe conocerse de qué datos se va disponer y cuáles son los resultados esperados.

Además, se ha de tener en cuenta que un buen algoritmo debe ser un método *general* para resolver todos los casos posibles del mismo problema.

Tiempo finito Los algoritmos deben finalizar.

Relacionados con el concepto de algoritmo, se tienen los siguientes:

Definición 1.2 (Entorno) *Conjunto de objetos necesarios para llevar a término una tarea.*

Como en toda disciplina, en programación se debe conocer qué objetos se pueden manejar y cómo se deben manejar. Este será el objetivo principal del próximo tema.

Asociado al concepto de entorno, aparece el de *estado del entorno*, es decir, la descripción del estado en que se encuentran los objetos del entorno en un momento dado.

Un algoritmo debe actuar para que el entorno cambie progresivamente de estado. Así se irán obteniendo los resultados a partir de los datos.

Definición 1.3 (Acción) *Un suceso finito, con un efecto definido y previsto.*

De nuevo es preciso puntualizar las características descritas,

Finito La acción debe finalizar,

Efecto definido También denominado *determinismo*: las acciones tienen una interpretación única,

Efecto previsto Se deben conocer las consecuencias de las acciones.

Definición 1.4 (Proceso) *El resultado de ejecutar una o varias acciones.*

Definición 1.5 (Procesador) *La entidad que puede comprender y ejecutar de forma eficaz un algoritmo.*

Se han hecho ya muchas definiciones y aún no se ha dicho nada sobre los programas. Para distinguir este concepto, se cree conveniente describir antes cómo funciona un ordenador.

Estructura y Funcionamiento de un Computador. Programas.

En un computador se distinguen tres partes principales (fig. 1.1):

- Las Unidades de Entrada y/o Salida, que permiten la comunicación con el computador (introducir datos e instrucciones, visualizar resultados, etc.)
- La Unidad Central de Proceso, que es la que dirige el funcionamiento de la máquina, y
- La Memoria Central, que es donde se almacenan los datos y los resultados, así como las instrucciones. Se organiza internamente en *posiciones de memoria*, y en cada una de estas posiciones se almacena una cantidad determinada de información (una *palabra* de memoria).

¿Cómo funciona? La Unidad Central de Proceso tiene dos componentes principales: la Unidad de Control y la Unidad Aritmético-Lógica. La Unidad de Control conoce en todo momento en qué posición de memoria se encuentra la siguiente instrucción que debe ejecutar el computador y se encarga de que se ejecute. Primero recoge de la Memoria Central la información necesaria para ejecutarla (la instrucción y los datos correspondientes); cuando tiene la instrucción, la interpreta para saber qué es lo que debe hacerse y la ejecuta. Si es una operación de tipo aritmético o lógico, se encarga de ordenar a la Unidad Aritmético-Lógica que la realice y le suministra los datos sobre los que debe operar. Finalmente, si se produjera algún resultado, la Unidad de Control lo almacena en la Memoria

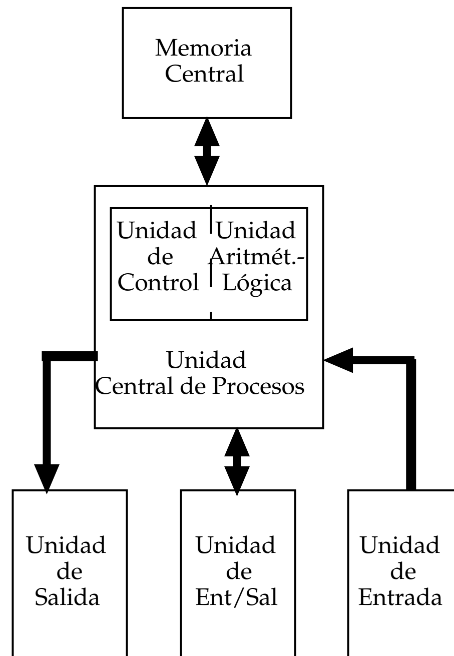


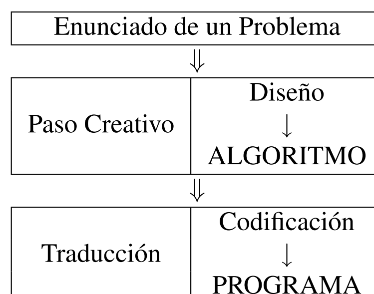
Figura 1.1: Estructura simplificada de un computador.

Central. Este proceso se repite desde la primera instrucción hasta la última, respetando el orden de la secuencia.

Esta máquina almacena los datos y las instrucciones en memoria en un determinado formato. Esto obligará a que los algoritmos deban *traducirse* a un lenguaje que el computador comprenda, a un *lenguaje de programación*. Por lo tanto,

> **Definición 1.6 (Programa)** *Es la codificación de un algoritmo en un lenguaje de programación.*

Una vez que se diseñado el algoritmo, la traducción de este a un lenguaje de programación para obtener un programa es muy directa. Basta con conocer el lenguaje de programación a utilizar y disponer de un *procesador del lenguaje* que permita generar el código ejecutable por el computador. Como ya se ha hecho antes, se puede establecer una analogía entre el esfuerzo intelectual de escribir una novela y el de escribir un algoritmo; la generación del programa a partir del algoritmo sería equivalente al esfuerzo de traducir una novela a otro idioma.



Hay dos tipos de procesadores de lenguaje, Intérpretes y Compiladores. En ambos casos se trata de programas que pueden traducir texto escrito en un lenguaje de programación concreto a instrucciones del *lenguaje máquina*, directamente ejecutables por el computador. Y es entonces cuando el computador comienza a trabajar tal y como se ha descrito anteriormente. Y, de nuevo, el proceso es automático: el único paso creativo en este proceso es el diseño del algoritmo; tanto la codificación a un lenguaje de programación como la traducción posterior a lenguaje máquina como la posterior ejecución no suponen esfuerzo creativo alguno. Por lo tanto, si se producen malos resultados al ejecutar un programa nunca es debido ni al computador ni al procesador del lenguaje. Los malos resultados son consecuencia de malos algoritmos.

En esta asignatura, no sólo se aprenderá a diseñar algoritmos, también se implementarán en diferentes lenguajes de programación para ejecutarlos sobre un computador.

Entre otros, hay dos tipos de paradigmas de programación: el *imperativo* y el *declarativo*. Este último, a su vez, se divide en el paradigma *lógico* y en el *funcional*.

En esta asignatura se utilizará la programación imperativa, que consiste en diseñar los algoritmos (por lo tanto, los programas) mediante una secuencia de instrucciones que definen cómo resolver un determinado problema (por lo tanto, es como si se diseñara una secuencia de "órdenes" que el computador debe seguir para obtener los resultados a partir de los datos). Por contra, en programación declarativa, los algoritmos son secuencias de instrucciones que definen qué problema hay que resolver, no cómo.

Además de seguir diferentes estilos de programación, es posible utilizar diferentes *metodologías de programación*. En esta asignatura se utilizará la *programación estructurada*, que se introducirá en los temas 2 y 3, para asegurar la legibilidad de los diseños realizados y una de las metodologías de diseño más simples, la denominada *TopDown* o de Refinamiento Progresivo, que se introducirá en el tema 4.

Hay lenguajes orientados hacia metodologías más concretas. Una de las más difundidas es la denominada *Programación Orientada a Objetos*, POO, que se caracteriza por el modo en que manipula la información: un *objeto* es una entidad lógica que contienen tanto datos como el código necesario para trabajar con esos datos. Otra metodología bastante difundida en la actualidad es la *Programación Orientada a Eventos*, POE, que se caracteriza por que el control de la ejecución de un programa queda relegado a la aparición de algún suceso externo al propio programa (la aparición de una interrupción, por ejemplo).

1.2. La Programación como Disciplina de Ingeniería.

La programación es una disciplina que, al igual que otras disciplinas de ingeniería, se fundamenta en una teoría, una metodología y un conjunto de técnicas de diseño.

Cada una de estas partes de la disciplina es consecuencia de la investigación y la experiencia adquirida en los últimos años y contribuye a hacer que la programación sea una tarea *eficaz*, es decir, que dé los resultados esperados, y *eficiente*, es decir, que necesite de un tiempo de desarrollo razonable.

La informática y, por lo tanto, la programación, son disciplinas jóvenes en relación a otras disciplinas de la ingeniería. De ahí, que un informático deba estar dispuesto a asumir el compromiso de una continua formación, a medida que se obtienen nuevos resultados en este campo.

Pero, además, debe asumir la gran contradicción que representa el ser especialista en un campo

que, por un lado, está muy presente en la vida diaria (la presencia de computadoras, así como el número de actividades diarias que se ven influidas por un proceso informático, es cada día mayor) pero, por otro, es “el gran desconocido”; en particular, el trato con clientes profanos en la materia suele ser especialmente díficil. El desconocimiento puede tener como consecuencia que se asocie la “magia” a lo que debe ser una disciplina de trabajo: se piden resultados, antes que una buena planificación.

Por ello es tan importante, de cara a la actividad profesional, el hecho de adquirir una buena base metodológica.

1.2.1. Etapas del Desarrollo del Software.

El proceso de desarrollo de programas, dentro del paradigma imperativo, se divide en varias etapas, que se enumeran a continuación:

1. Definición del Problema y Análisis de Requisitos,
2. Diseño del Algoritmo,
3. Implementación,
4. Pruebas,
5. Operación, Mejoras y Mantenimiento.

La última fase se realiza realmente durante la vida operativa del programa diseñado; si fuera necesario hacer mejoras o cambios, se habría de retroceder a etapas previas del desarrollo.

Lo deseable es que estos pasos se ejecuten en secuencia, es decir, que no comience una etapa hasta que no haya finalizado la anterior. Esta sería la consecuencia de aplicar una buena metodología de trabajo, ya que supondría la forma más eficaz y eficiente de trabajar. Y actualmente se dispone de los conocimientos necesarios para poder desarrollar así el trabajo.

Sin embargo, todavía es normal que este desarrollo se vea interrumpido por continuas realimentaciones, mucho antes de llegar a la última etapa: clientes que cambian la definición del problema, errores detectados en la etapa de prueba como consecuencia de un mal diseño del algoritmo... Una buena medida sobre la calidad y la profesionalidad del programador, o del equipo de programadores, puede ser, precisamente, el número de realimentaciones que deba realizar en el esquema anterior.

A continuación se describe cada una de estas etapas:

Definición del problema y análisis de requisitos: Un cliente plantea un problema que necesita una solución informática mediante la construcción de un sistema informático; en este sistema pueden integrarse varios equipos, distintos programas y bases de datos. El primer paso consiste en un análisis del problema y es un paso muy laborioso, bien sea por el nivel de comprensión del problema, bien por el planteamiento que haya realizado el cliente. Una vez que se ha definido el sistema (qué partes, cuáles son, qué programas hay que desarrollar, cómo se deben integrar, cómo deben colaborar), se debe realizar un análisis minucioso de los requisitos que supone la definición informática del mismo.

El resultado de esta etapa debe ser un enunciado preciso y claro del problema.

Esta etapa entra dentro de la disciplina denominada *Ingeniería del software* y queda fuera de esta asignatura. Por lo tanto, nosotros partiremos siempre de un enunciado ya definido, a partir del cual se desarrollarán las etapas posteriores.

Diseño del algoritmo: En esta etapa se debe realizar un diseño que permita obtener la solución deseada al problema.

Esta es la etapa más importante y costosa del proceso; de su éxito depende, en buena medida, el coste y el éxito de las etapas posteriores.

Esta etapa es el objetivo básico de esta asignatura.

Implementación del programa: Para ejecutar el algoritmo hay que traducirlo a un lenguaje de programación. Esta etapa es, pues, una etapa poco costosa y sencilla ya que se trata de traducir el diseño realizado en la etapa anterior.

Esta etapa es el segundo objetivo de la asignatura.

Pruebas: En esta etapa se trata de probar el programa resultante con diferentes datos de entrada que reciben el nombre de *juegos de prueba*.

Es importante saber que los juegos de prueba sólo sirven para comprobar que el programa *no es correcto*, cuando para algún juego no se obtienen los resultados previstos. *Pero no sirven para comprobar que el programa es correcto* (a no ser que se hagan juegos que permitan comprobar *todas* las entradas posibles, lo que es prácticamente imposible). Para comprobar que un programa es correcto hay que utilizar las *Técnicas de Verificación Formal*.

Además, es una etapa que suele realizarse mal, a no ser que se adquieran técnicas y pautas que permitan elegir convenientemente los juegos de prueba. Suele verse como una etapa aburrida, lo que puede provocar que no se verifiquen casos de entradas de gran tamaño, o que no se comprueben casos extraños o de condiciones extremas de los datos.

Por lo tanto, el éxito en esta etapa viene dado, primero, por el hecho de haber realizado un buen diseño del algoritmo, y segundo, por haber realizado una buena implementación y haber realizado un buen conjunto de juegos de prueba.

Operación, mejora y mantenimiento: Una vez que se ha obtenido y probado el programa, no ha terminado el proceso. Un programa ya acabado puede cambiar, bien porque se detecten fallos durante su operatividad que no se habían detectado previamente, bien porque sea necesario mejorar o adaptar algunas de sus prestaciones. Realizar este cambio supone reproducir todo el proceso desde el inicio.

En esta etapa es donde cobra mayor importancia que el diseño del programa se haya realizado utilizando un buen estilo de programación. Un programa se escribe una vez, pero se lee muchas veces. Y, posiblemente, por parte de un equipo ajeno a su diseño. Ello hace necesario un buen estilo, *legible* y, además, pone de relieve la importancia de realizar una buena *documentación*, a ser posible en todas las etapas de desarrollo, de forma que se ofrezca información significativa, no superflua, de cada una de las etapas. Especialmente de la de diseño.

Se podría hacer un resumen, un tanto cruel, del proceso: el computador es tonto, pero es muy rápido y puede desarrollar de forma precisa todas las acciones que se le ordenen; el programador es inteligente, es creativo y debe ser preciso en la definición de un proceso. Y el cliente es ... impredecible y exigente.

1.2.2. Objetivos de la Asignatura.

La asignatura se centrará básicamente en el diseño de algoritmos y en su posterior implementación en lenguajes de programación imperativos.

A partir de un enunciado concreto, se deberá hacer el diseño de un algoritmo. Puede haber más de un algoritmo que permita resolver el mismo problema. Por lo tanto, se han de tener criterios para escoger el algoritmo más adecuado para la implementación.

Los algoritmos diseñados, han de tener las siguientes características,

Correctos harán lo que deben,

Inteligibles fáciles de entender y de leer,

Eficientes lo que tengan que hacer, lo harán lo más rápidamente posible,

Generales con pocos cambios, deben poderse adaptar a problemas similares al que resuelven.

En el diseño de algoritmos, se seguirán las siguientes etapas:

1. *Entender el problema*: Lo que no se entiende, no se puede solucionar. La especificación formal del problema puede ser una ayuda. En esta asignatura se usarán dos herramientas: primero, identificar en cada problema cuáles son los datos y cuáles son los resultados que se espera conseguir. Y, segundo, establecer las condiciones iniciales, que también se denominan *precondiciones* para poder establecer el objetivo u objetivos a conseguir con el proceso, también denominados *postcondiciones*.
2. *Plantear y planificar la solución*: No hay recetas mágicas en este paso, es el paso creativo. Pero sí que es deseable intentar no "reinventar la rueda", utilizando en la medida de lo posible esquemas generales que permitan resolver problemas de naturaleza similar y, además, utilizar *bibliotecas de programas*. Esto, junto con la metodología de diseño descendente, permite abordar la resolución de problemas complejos de forma eficiente.
3. *Formular la solución*: Plasmar la solución diseñada dentro de un estilo de programación concreto. En este caso, la programación imperativa. Y, además, reformulando la solución en términos de programación.
4. *Evaluar la corrección*: Aún cuando no sea el objetivo de la asignatura utilizar herramientas de verificación formal, es deseable que se adquiera la costumbre de razonar sobre el algoritmo que se haya diseñado.
5. *Implementar la solución*: Es preciso adaptarse al proceso que supone traducir los diseños. El proceso debe ser automático pero, al igual que ocurre cuando se aprende un nuevo idioma, puede haber matices sintácticos que dificulten la tarea la principio. Además, dependiendo de que se trabaje con un intérprete o con un compilador, se debe realizar una adaptación al entorno. En concreto, en el caso del compilador, se debe manejar también un *montador de programas*.

   Otro punto a tener en cuenta es la conveniencia de realizar buenas pruebas: aunque ya se ha dicho que no demuestran la corrección, pueden ayudar a depurar los programas realizados. Y, además, también es importante adquirir el hábito de documentar el trabajo realizado.

1.3. Resumen.

El objetivo general de esta asignatura es aprender a diseñar algoritmos de mediana complejidad y a implementarlos como programas dentro del paradigma imperativo, utilizando la programación estructurada.

Debemos entender la programación como una disciplina de la ingeniería; por ello, nos impondremos los objetivos descritos en la sección 1.2.2. Resultaría una buena idea que los releyeráis a medida que avanza el curso para no perderlos de vista; además, a medida que avancemos en la materia los entenderéis mejor.

1.4. Glosario.

algoritmo Conjunto explícito de reglas para resolver un problema en tiempo finito.

codificación Traducción de un algoritmo a un lenguaje de programación.

computador Autómata de cálculo gobernado por medio de programas.

entorno Conjunto de objetos necesarios para llevar a término una tarea.

especificación Formalización del enunciado de un problema.

estado Descripción del entorno en un momento dado.

implementación Realización física de un proyecto.

lenguaje algorítmico Lenguaje artificial que permite expresar sin ambigüedades un algoritmo. En nuestro caso, un dialecto del lenguaje C que nos permitirá expresar algoritmos sin tener que sufrir las iras del compilador.

lenguaje natural Cualquier lenguaje que se utiliza como forma natural de comunicación: castellano, catalán, inglés, el lenguaje de signos, etc.

lenguaje de programación lenguaje creado para expresar programas y que puede ser interpretado por un computador.

programa Codificación de un algoritmo en un lenguaje de programación.

sintaxis Reglas de formación de un lenguaje.

semántica Significado de los símbolos de un lenguaje.

1.5. Bibliografía.

1. "Fonaments de Programació I", M.J. Marco, J. Álvarez & J. Villaplana. Universitat Oberta de Catalunya, Setembre 2001.

2. "Introducción a la Programación (vol. I)", Biondi & Clavel. Ed. Masson. 1991.

3. "La crisis crónica de la programación", W. Wayt Gibbs. Investigación y Ciencia, Noviembre 1994. Pág. 72 – 81.

   Como podéis ver, este artículo es de 1994; uno esperaría (o por lo menos desearía) que ya se hubiera quedado anticuado, pero ... el siguiente que os recomendamos es de este mismo año y el título lo dice todo; es decir, seguimos igual (esperemos que no peor, aunque nunca se sabe).

4. "Why software is so bad", Charles C. Mann. Technology Review, July/August 2002.

Capítulo 2

Conceptos Básicos

Índice General

2.1. Variables. Tipos Básicos y Expresiones. **11**
2.2. C: Estructura Básica de un Programa, Variables y Tipos Básicos. **17**
2.2.1. Ejemplo de Programa en C. 17
2.2.2. Tratamiento de Variables y Tipos Básicos. 19
2.2.3. Entrada/Salida de Datos. 24
2.3. Resumen y Consideraciones de Estilo. **28**
2.4. Glosario. **28**
2.5. Bibliografía. **29**
2.6. Actividades y Problemas Propuestos. **29**

"Un programa hace lo que se le ordena hacer, no lo que se quiere que haga."
Anónimo.

2.1. Variables. Tipos Básicos y Expresiones.

Se ha definido el *entorno* como el conjunto de objetos que se utilizan en el desarrollo de un trabajo. Cualquier trabajo supone un conocimiento del entorno, de saber qué herramientas se pueden utilizar y cómo utilizarlas. También en el diseño de algoritmos y programas el primer paso consistirá en la familiarización con los objetos a manejar.

La informática es la ciencia para el proceso automático de la información. Por lo tanto, los objetos a manejar en un entorno de programación serán items de información, valores a procesar. Estos objetos quedarán definidos por tres atributos:

Nombre que permite identificar a cada objeto de forma única,

Tipo que permite saber para qué sirve un objeto, y

Valor que indica cuál es la información almacenada en un objeto en un momento dado.

A continuación, se propone un ejemplo que permitirá ilustrar algunos de los conceptos referidos a cómo manipular, representar y procesar los objetos que forman parte del entorno de la programación.

Cálculo de la media aritmética de 4 valores enteros:

Este problema es muy fácil de definir. Es preciso que nos den los 4 números enteros y el resultado se obtiene directamente de un proceso muy simple: basta con sumar los 4 valores y dividir el resultado entre 4,

$$media = \frac{num1 + num2 + num3 + num4}{4} \quad .$$

La expresión anterior se puede cosiderar un diseño de un algoritmo, ya que indica de forma precisa y sin ambigüedades el proceso a realizar. Además, es una expresión general, ya que se ha dejado el proceso en función de valores genéricos de forma que basta con instanciar num1*,* num2*,* num3 *y* num4 *con 4 valores concretos y se puede obtener la solución.*

Sin embargo, dicha expresión matemática no sería completamente válida en programación, ya que la sintaxis de un lenguaje de programación obliga a escribir las acciones en una única línea. Una notación habitual es la siguiente,

```
media = (num1 + num2 + num3 + num4) / 4 ;
```

que es un algoritmo completamente válido.

Aún se puede trabajar un poco más con este ejemplo. Si se reflexiona sobre cuál es la forma de resolver la expresión anterior (la suma es una operación binaria y cuando hay más de dos operandos se aplica la operación asociativa, tanto si el procesador es una persona como si es una CPU), se llega a la conclusión de que el proceso anterior se podría haber descrito mediante el siguiente algoritmo:

```
suma = num1 + num2 ;
suma = suma + num3 ;
suma = suma + num4 ;
media = suma / 4 ;
```

Esta segunda formulación presenta una ventaja sobre la anterior en el sentido de que además de ser general, también es más fácilmente generalizable. Para ello, considérese cómo se expresaría en ambos casos la solución al problema de calcular la media de 100, o 1000, números enteros.

En el ejemplo anterior se manejan siete objetos. Cada uno se ha identificado con un *nombre*. En primera instancia se pueden distinguir dos categorías:

- Los objetos `num1`, `num2`, `num3` y `num4` son imprescindibles ya que son los *datos*; si no se les da un valor concreto, no se puede desarrollar un cálculo efectivo. Y el objeto `media` también es imprescindible: es el *resultado* esperado y si no se da a conocer al final del proceso, éste habrá sido inútil.

  A estos objetos se les denomina *parámetros* y viene impuestos por la definición del problema: son necesarios en cualquier formulación del algoritmo, como se puede observar al comparar las dos versiones propuestas en el ejemplo.

  En el tema 4 se introducirá el concepto formalmente. Por ahora basta con la idea intuitiva de que es preciso que el programa tenga posibilidad de comunicarse con el exterior, de dar valores concretos a los datos para así poder emitir unos resultados.

- Los objetos `suma` y `4` son necesarios para desarrollar los cálculos; por ejemplo, `suma` se asocia a la obtención de resultados intermedios que es preciso conservar para llegar a un resultado final correcto.

  Hay una diferencia muy importante entre ellos: a medida que se desarrolle el cálculo, el objeto `suma` cambiará de valor, es un objeto de valor *variable*, mientras que el `4` es un objeto de valor *constante*.

Variables.

Del ejemplo anterior se desprende que hay objetos que, según el atributo *valor*, son constantes o variables. En el desarrollo de los algoritmos, se invierte la mayor parte del esfuerzo en manipular correctamente objetos como `suma`, es decir, variables. No resulta exagerado decir que el principal objetivo de un programador será saber cuántos objetos necesita y cómo manipularlos, para poder desarrollar correctamente el algoritmo: los datos y resultados deben quedar especificados al definir el problema, pero el número y uso correcto de variables forman parte del diseño de la solución. De ahí, la importancia que adquieren en la definición del entorno de trabajo.

Tipos.

No todos los objetos del ejemplo son del mismo tipo. Se manejan valores enteros, los de `num1`, `num2`, `num3`, `num4`, `4` y `suma`, pero el resultado del cálculo, `media`, será un valor real.

El atributo *tipo* permite clasificar los objetos según su utilidad y las operaciones que se pueden hacer con ellos.

Un tipo se define indicando el conjunto de valores que lo forman. Esto puede hacerse por comprensión - *enunciando una propiedad que cumplen todos los objetos de ese tipo* - o bien por extensión - *enumerando todos los posibles objetos de un tipo dado*.

Los tipos se pueden dividir en **básicos** y **estructurados**[1]. Hay cuatro tipos básicos:

Tipo ENTERO: Es útil cuando se quiere representar información cuyo valor sólo puede ser un valor entero. Por ejemplo, el número de personas matriculadas en una asignatura, el número de volúmenes de una biblioteca o el número de votos que recibe una candidatura política.

En principio, el tipo entero se puede asimilar al conjunto $\mathcal{Z}$ de los enteros. Sus valores se representan como se hace normalmente en Matemáticas, con o sin signo, dependiendo de que sean positivos o negativos: 34, -16, 8456, 0, 144,...

Pero, a la hora de diseñar un algoritmo, especialmente cuando se convierta en un programa, se debe tener en cuenta que en programación el tipo entero debe definirse como un subconjunto de los números enteros, puesto que el conjunto $\mathcal{Z}$ es infinito y la capacidad de almacén de un computador es finita: se habla entonces del *rango* de los números enteros en una determinada máquina y/o lenguaje de programación; es usual utilizar una constante especial, `maxentero`, que indica el máximo entero en valor absoluto que es capaz de representar la máquina o el lenguaje. En general, en el entorno de programación se define el tipo entero como el subconjunto

$$\{x \in \mathcal{Z} \mid -maxentero \leq x < maxentero\}.$$

Tipo REAL: La necesidad de manejar valores reales surge en cuanto se quiere manipular información numérica cuyo valor puede ser fraccionario como, por ejemplo, la nota media de acceso

[1] Se introducirán en el tema 5

de una determinada persona a los estudios universitarios, la probabilidad de que un determinado libro esté disponible en una biblioteca o el porcentaje de votos conseguido por una partido político en unas elecciones.

La presentación de este tipo es muy similar a la del tipo entero. El tipo real, en principio, se puede asimilar al conjunto $\mathcal{R}$ de los reales y sus valores se representan siguiendo la notación matemática habitual: con o sin signo, dependiendo de que sean positivos o negativos, con puntos decimales o con notación científica (mantisa y exponente): 34.23, -16.5E2, 84.156, 0.7E-3, -44.58,...

Al igual que con el tipo entero se debe tener en cuenta que hay un *rango* de representación y, además, introducir un nuevo concepto: el de *precisión*, ya que el conjunto $\mathcal{R}$ es un infinito de infinitos, es decir, entre dos números reales cualesquiera hay infinitos números reales. Si el concepto de rango está relacionado con el mayor número que la máquina es capaz de representar, el concepto de precisión está relacionado con el número más pequeño que puede distinguir del cero. Por ejemplo, con precisión 4, el número más pequeño que se puede representar es el 0.0001, ya que el 0.000099 se tomaría como 0 al no representar la máquina más allá de cuatro dígitos después del punto decimal[2] . En general, en el entorno de programación se define el tipo real como el subconjunto de los reales menores en valor absoluto que una cierta constante `maxreal` y tales que son distinguibles con una precisión dada.

Tipo CADENA: Es necesario para manipular información como el nombre de una persona o el título de un libro; en este tipo de información se engloba lo que se suele llamar “proceso alfanumérico”, ya que además de los ejemplos indicados entrarían en este tipo información como una dirección postal, que engloba tanto letras como números (y, en este caso, el proceso de los números no supone un cálculo, sino la identificación de un código postal o un determinado piso).

En algorítmica suele considerarse que cualquier objeto formado como una combinación de cualquier longitud de carácteres de un conjunto básico es de tipo cadena. Sin embargo, hay que tener en cuenta que no todos los lenguajes de programación tienen el mismo comportamiento con este tipo: algunos siguen esta misma definición (python, por ejemplo) y otros, como el `C`, distinguen entre un único carácter, que es el tipo básico, y una cadena, que es un tipo estructurado[3] .

Como conjunto básico de carácteres se suele aceptar como estándar la tabla ASCII. En ella se recogen todas las letras de los alfabetos occidentales, tanto en mayúsculas como en minúsculas, los dígitos y un conjunto de caracteres como símbolos de puntuación, ortográficos, o especiales como ‘@’, ‘ %’, ‘&’,... Las cadenas se escribirán entre comillas (“). Por ejemplo, “A”, “Programación”, “““ (la cadena formada por una comilla), “” (la cadena vacía, es decir, la de longitud cero), “ ” (el carácter blanco[4]), “12538” (una cadena formada por dígitos),...

Tipo BOOLE o LÓGICO: es el formado por los valores {`falso`, `cierto`} y que son básicos cuando el proceso de los datos supone la evaluación de condiciones.

Todos los lenguajes recogen este tipo bien sea de forma explícita, con una definición similar a la anterior, o implícita, normalmente basada en el tipo entero (es habitual encontrarse el criterio ’0’ `falso`, ’1’ (o cualquier valor distinto de 0) `cierto`, aunque también podría admitirse cualquier criterio de dualidad como positivo `cierto`, negativo `falso`, por ejemplo).

[2]En realidad, el concepto de precisión requiere una explicación más compleja, porque para entenderlo bien es preciso estar familiarizado con la representación “exponente-mantisa”. Lo que se ha dicho es cierto para números pequeños, ya que la precisión con que se representa un valor es mayor cuanto más cerca está ese valor del cero. Los números muy grandes suelen representarse con poca precisión. Se volverá a discutir el tema más adelante.

[3]Para evitar confusiones, y ya que en prácticas se trabajará en `C` la mayor parte del tiempo, se ha optado por seguir su sintaxis: apóstrofes (`'c'`) para carácteres y comillas (`"cadena"`) para cadenas.

[4]Para evitar confusiones entre la cadena vacía y la formada por un carácter blanco, será habitual utilizar el símbolo $\flat$ para representarlo. Por lo tanto, el ejemplo quedaría de la siguiente forma,“$\flat$”.

La asignación.

Si el entorno de trabajo de un algoritmo está formado por valores contenidos en objetos, las acciones que realice un algoritmo por fuerza deben estar relacionadas con lo que se pueda hacer para cambiar el valor de dichos objetos.

Sólo hay una acción que pueda realizar el procesador para manipular valores. Es la *asignación*, la acción de dar un valor a un objeto. Es, una *acción destructiva*, ya que en el momento en que se da un valor a un objeto variable, se pierde el valor anterior.

Definición 2.1 (Asignación) *Acción destructiva que permite dar un valor a un objeto variable. La notación que se utilizará para denotarla es la siguiente,*

`<variable>=<expresión>`,

donde `<variable>` *siempre será el nombre de una variable, = representa la acción de asignar y* `<expresión>` *el valor a asignar. Este valor será del mismo tipo que la variable y puede ser,*

1. *un valor constante (el valor de un objeto constante),*
2. *el valor de otra variable (indicando el nombre de dicha variable),*
3. *el valor resultante de calcular una expresión (después de realizar el cálculo indicado).*

En cualquier caso, siempre se evalúa primero el valor de `<expresión>` *y después de obtenido se realiza la asignación sobre* `<variable>`.

Expresiones.

Definición 2.2 (Expresión) *Combinación de operandos y operadores, del mismo tipo, que expresan un cálculo que se ha de evaluar para obtener un resultado.*

Tanto en la definición de *asignación* como en la de *expresión*, se hace referencia al concepto de *tipo*.

En la de la asignación se destaca lo que se conoce como *coherencia de tipos*: una asignación sólo es válida si `<variable>` y `<expresión>` son del *mismo tipo*. Es decir, a una variable de tipo entero, por ejemplo, se le puede dar el valor `3`, pero no `'A'`, `4.5` o `cierto`. A una variable de tipo real se le puede asignar el valor de otra variable de tipo real, pero no el valor de una variable carácter, por ejemplo. Algunos lenguajes, no todos, admiten la *coerción* entre los tipos entero y real; es decir, puesto que los valores enteros forman un subconjunto de los valores reales se admite el uso de valores enteros y reales en una misma expresión, pero sabiendo que el resultado será de tipo real y sólo se le podrá asignar a una variable de tipo real.

De ahí, que en la definición de la expresión, también haya que hacer referencia al tipo, ya que del concepto de coherencia, se sigue que también las expresiones tienen tipo. Una expresión no es más que una combinación de objetos (operandos) de un mismo tipo y operadores asociados a ese tipo, que expresan un cálculo. El tipo de la expresión es el de sus operandos y operadores. Puesto que se han definido cuatro tipos básicos de objetos, también se distinguirán cuatro tipos básicos de

operadores,

Operadores ENTEROS: Se aceptará cualquier operador de la aritmética entera. Los más típicos son – (cambio de signo) entre los operadores unarios y, entre los binarios, los siguientes: + (suma entera), – (resta entera),* (multiplicación entera), el cociente de la división entera y el resto de la división entera.

Estos operadores tienen implícita una jerarquía: de mayor a menor, cambio de signo, la multiplicación y la división y, por último, la suma y la resta. De entre varios operadores con la misma prioridad, se realiza primero la operación que se lee primero de izquierda a derecha. Esta prioridad se puede modificar utilizando paréntesis. Por ejemplo, 3+4*5, daría como resultado 23, mientras que (3+4)*5, daría 35.

Operadores REALES: Serán los usuales en la aritmética real. Se suele disponer también del cambio de signo y de los binarios más típicos: + (suma real), – (resta real),* (multiplicación real), / (división real) y la potenciación.

La jerarquía es similar: primero cambio de signo, la potenciación, después la multiplicación y la división y, por último, la suma y la resta. También en este caso se recurre al uso de paréntesis si se desea modificar la prioridad o, simplemente, para mejorar la legibilidad de la expresión.

Operadores de tipo CADENA: Permiten realizar operaciones con cadenas. Uno de lo más típicos es la concatenación, que se suele denotar con el operador + (ver pie de página[5]). Dependiendo del lenguaje de programación, es habitual encontrar operaciones que permiten determinar la longitud de una cadena, o acceder a alguno de los caracteres individuales de una cadena, por ejemplo.

Operadores de tipo BOOLE: Hay dos clases de operadores boole. Por un lado están aquellos operadores que expresan operaciones del álgebra de Boole (*conectores*). Los más típicos son (`x` e `y` representan valores de tipo booleano):

- **AND**: `x AND y = cierto`, si y sólo si el valor de `x` es `cierto` y el valor de `y` es `cierto`. En cualquier otro caso, el resultado es `falso`.
- **OR**: `x OR y = falso`, si y sólo si el valor de `x` es `falso` y el valor de `y` es `falso`. En cualquier otro caso, el resultado es `cierto`.
- **NOT**: `NOT x = cierto`, si el valor de `x` es `falso`, `NOT x = falso`, si el valor de `x` es `cierto`.

Pero, además, existen los llamados *operadores de relación*, que se caracterizan por relacionar objetos del mismo tipo en una expresión cuyo resultado es lógico. Esa relación es una relación de orden, por lo tanto los operadores de relación son,

$$=, <, \leq, >, \geq, \neq.$$

Entre objetos reales y enteros, la relación de orden se define como en $\mathcal{R}$ y $\mathcal{Z}$, respectivamente. Entre objetos de tipo carácter, se adopta el orden de la tabla ASCII (que coincide con el lexicográfico en el caso de las letras del alfabeto, salvando que las letras mayúsculas son anteriores a las letras minúsculas). Entre objetos de tipo `BOOLE`, se suele aceptar que `falso` es menor que `cierto`.

[5]Como se ha visto, con el mismo símbolo se pueden representar distintas operaciones. En el caso del símbolo +, puede representar la suma entera, la suma real y la concatenación. Esta propiedad se conoce como polisemia.

2.2. El Lenguaje C: Estructura Básica de un Programa, Tratamiento de Variables y Tipos Básicos.

En esta subsección se verá cómo adaptar el tratamiento de variables y tipos cuando el lenguaje de programación es C. Entre los objetivos de la asignatura no se encuentra el de formar expertos programadores en lenguaje C, ni en ningún otro lenguaje. La asignatura pretende introducir al alumno en el mundo de la algorítmica y la programación, y en este sentido hay que entender que los lenguajes de programación son una herramienta y no un fin. Ni esta sección, ni las equivalentes en temas posteriores, pretenden ser una guía exhaustiva para aprender el lenguaje de programación C; la idea es ayudar a aquéllos que, teniendo nociones básicas de programación, necesitan conocer y utilizar dicho lenguaje de programación.

2.2.1. Ejemplo de Programa en C.

A partir del algoritmo diseñado como ejemplo en la sección anterior,

```
suma = num1 + num2 ;
suma = suma + num3 ;
suma = suma + num4 ;
media = suma / 4 ;
```

se construirá el primer programa en C, que es el siguiente[6]:

```
1  int main()
2  {
3          /* Datos */
4          int num1, num2, num3, num4;
5          /* Resultados */
6          float media;
7
8          int suma;
9
10         suma = num1 + num2 ;
11         suma = suma + num3 ;
12         suma = suma + num4 ;
13         media = suma / 4.0 ;
14 }
```

Lo primero que se puede observar es que ha aumentado el número de líneas; a continuación, se comentará qué diferencias hay y por qué:

- Línea 1: `int main()` define lo que se conoce como la *función principal* de un programa en lenguaje C. Esta función siempre debe existir y es la primera que se ejecuta.

  Para definir una función en C se indica, en primer lugar, el tipo de datos que devuelve. En este caso, el estándar ANSI obliga a que la *función principal* devuelva necesariamente un valor

[6] Los números de las líneas, en este y otros ejemplo, sólo se muestran por claridad y no se deben escribir en un programa en C.

entero y eso es lo que indica la palabra reservada `int` (de **int**eger en inglés). A continuación, se indica el nombre de la función; en este caso `main`, y también por indicación del estándar. Por último se debe indicar, entre paréntesis, los parámetros de la función, si los tiene. Al utilizar `()` se indica que la función no tiene ningún parámetro. Aquí el estándar permite distintas opciones, pero por el momento se utilizará ésta.

- Líneas 2 y 14: las llaves ("{" y "}") sirven en `C` para indicar el inicio y el fin de un bloque de instrucciones. En el ejemplo, el bloque formado por las instrucciones que forman la función principal.
- Líneas 3 y 5: se han utilizado *comentarios* para indicar cuáles son los datos y cuál es el resultado.

  Un comentario es una o varias líneas que no forman parte del programa (de hecho, el procesador del lenguaje los ignora), sino que ayudan a leerlo y entenderlo. Utilizarlos bien es el primer paso de un programa legible y bien documentado. En `C` se pueden poner comentarios en cualquier parte del programa utilizando "`/*`" para indicar el inicio de un comentario y "`*/`" para indicar el final.
- Líneas 4, 6 y 8: Se está realizando la *definición de variables*. La definición de variables se realiza al inicio de una función indicando el tipo de la variable y su nombre. Con `int` se indica que una variable es de tipo entero (para los reales, se utiliza `float`). Para cada variable se indica qué nombre la identifica. La definición de variables de cada tipo termina con el símbolo `;` (punto y coma).

  Por convenio, en la asignatura se definirán primero los datos, después los resultados y, por último, las variables propias del algoritmo.
- En las líneas de la 10 a la 13, figuran las instrucciones que se utilizaron para realizar el primer algoritmo ejemplo. En ellas, se utilizan asignaciones, =, y expresiones tal y como se definían en la sección anterior. De nuevo, se utiliza el punto y coma para finalizar una instrucción.

De acuerdo a este ejemplo, se podría hacer un primer esquema básico de cuales son las partes que forman un programa en `C`:

```
/* Función principal*/
int main() {
    /* Especificación de datos */
    /* Especificación de resultados */

    /* Declaración de variables de la función main*/

    /* Cuerpo del algoritmo */
}
```

A lo largo de este capítulo se irán viendo nuevos conceptos que completarán este primer esquema de un programa en `C`.

2.2.2. Tratamiento de Variables y Tipos Básicos.

Variables.

En la estructura básica de un programa, se observa que lo primero que se ha de hacer es la declaración de las variables. En `C`, antes de que se utilice una variable ha de ser declarada. Se declaran al principio de la función y son locales a ella: se crean al iniciar la función y se destruyen al finalizarla. Ninguna otra función puede acceder a sus valores.

La sintaxis para declarar una variable es la siguiente:

<**tipoDeDatos**> <**nombreVariable**>**;**

donde `tipoDeDatos` indica el tipo de datos de la variable que se quiere definir y `nombreVariable` es el nombre (o *identificador*) de la variable.

Es posible declarar varias variables a la vez si son del mismo tipo:

<**tipoDeDatos**> <**nombreVar_1**>**,** <**nombreVar_2**>**, . . . ,** <**nombreVar_n**>**;**

El lenguaje `C` impone las siguientes restricciones para formar el nombre de una variable: obligatoriamente ha de empezar por una letra, pudiendo ser el resto de carácteres del nombre letras, números o el símbolo de subrayado (_). No se pueden utilizar símbolos especiales o palabras reservadas del lenguaje. Hay que hacer notar que el `C` distingue entre mayúsculas y minúsculas.

Ejemplo de identificadores válidos:

```
a,  i,  j,  edad1, Peso1, Peso2, cont_1, mi_nombre, ...
```

`Peso1` es distinto identificador que `peso1`.

Ejemplo de identificadores no válidos:

```
2edad, Peso-1, año, 18deDic, main, mi#nombre...
```

Ejemplo de declaraciones válidas:

```
int a;
int  i, j, edad1;
float Peso1, Peso2;
```

Constantes.

La declaración de objetos de valor constante no es obligatoria en `C`, pero es útil por tres motivos principales:

- **Mejoran la claridad de los programas**, puesto que son fácilmente identificables y hacen que las expresiones sean más fáciles de leer. Es más significativo leer `2*PI*r` (habiendo asociado antes a `PI` el valor `3.1415927`) que `2*3.1415927*r`.
- **Evitan la introducción de errores**, ya que se define su valor una vez y después se utiliza su nombre simbólico. Considérese, por ejemplo, una constante numérica. Si cada vez que se ha de escribir se han de poner todos los dígitos que la forman, puede ocurrir que se introduzcan errores involuntariamente, siendo además estos errores difíciles de localizar. Siguiendo con el ejemplo del valor de `PI`, podría ocurrir que en algún sitio, por error, se escribiera involuntariamente `3.1425927`.
- **Permiten modificar el valor de una constante de forma sencilla**. El valor de una constante puede cambiar con el tiempo (por ejemplo, si una constante define el valor del IVA y éste se modifica por algún motivo, o si se quiere aumentar la precisión de un valor real). En estos casos, se hace necesario modificar el valor constante allá donde aparece (si no se ha definido como tal). Si se ha definido una constante para dicho valor, sólo hay que modificar la definición.

En `C` una constante se puede definir utilizando la directiva `#define` del preprocesador[7] que simplemente sustituye todas las apariciones de la constante en el programa por su valor:

#define <CONSTANTE> <valor>

También se pueden definir objetos constantes tal y como se hace para las variables pero anteponiéndoles el calificador `const` al tipo de datos y asignándoles un valor:

const <tipoDeValor> <CONSTANTE>=<valor>;

Ejemplos:

```
#define PI  3.1415927

const int TAM=200;
const float IVA=16.0;
```

Por claridad y para distinguirlas fácilmente de las variables, se suele poner el nombre de las constantes en mayúsculas como se muestra en los ejemplos.

Tipos de Datos Básicos.

El lenguaje `C` dispone de varios tipos de datos. El objetivo de esta subsección es comentar cómo utilizar en este lenguaje los tipos que se han definido anteriormente como básicos:

- El tipo *entero* se define utilizando la palabra reservada `int`. El tipo `int` puede ir acompañado de los calificadores `short` para enteros pequeños o `long` para enteros grandes. Los rangos de

[7]El preprocesador es un programa residente que se ejecuta antes de la compilación y que realiza determinadas acciones definidas por las directivas incluidas en nuestros programas, que empiezan por #.

valores para cada uno de los calificadores dependen del ordenador y el compilador utilizados. A efectos de esta asignatura se utilizará `int` sin calificadores.

- El tipo *real* se define utilizando la palabra reservada `float` o `double` si se quiere trabajar con una mayor precisión o con un rango mayor. El tipo `double` puede también ir acompañado del calificador `long` si se quiere trabajar todavía con una mayor precisión o con un rango mayor.
- El tipo *carácter* se define utilizando la palabra reservada `char` y sirve para almacenar un único carácter. Realmente el lenguaje `C` almacena en las variables de tipo `char` un valor entero entre 0 y 255, el correspondiente al código ASCII del carácter que se quiere almacenar. Más adelante, en el tema 5, se comentará cómo trabajar con *cadenas* (`string`) en `C`.
- El tipo *lógico* o *boole* no existe como tal en `C`. Se puede "simular" utilizando el tipo `char`, por ejemplo, pero es más habitual utilizar el tipo `int`, de forma que el valor 0 equivale a `FALSO` y cualquier otro valor a `CIERTO`.

  Sin embargo, una solución más elegante es utilizar un tipo enumerado (`enum`) tal y como se indica en el siguiente apartado.

Tipos enumerados.

Un *tipo enumerado* define un nuevo tipo cuyos objetos sólo pueden tomar los valores **constantes** definidos en la enumeración. En el lenguaje `C` se define utilizando la palabra reservada `typedef` del siguiente modo:

```
typedef enum {<v1>, <v2>, <v3>, ..., <vn>} <nombreDelTipo>;
```

Dos ejemplos típicos son los siguientes:

```
typedef enum {lunes,martes,miercoles,jueves,viernes,sabado,domingo} tDias;

typedef enum {FALSO,CIERTO} boole;
```

De esta forma, se ha definido el tipo `tDias` (cuyos objetos pueden tomar los valores `lunes, martes, ..., domingo`) y el tipo `boole` cuyos objetos pueden tomar los valores `FALSO` y `CIERTO`.

Esta última definición permitirá que se puedan manejar más cómodamente los objetos de tipo lógico. En el tipo `boole`, es muy importante el orden en que se definen los valores, ya que en un enumerado el lenguaje `C` asigna al primer valor, el valor 0, al segundo el 1 y así, sucesivamente. Siguiendo el orden anterior, se asegura que el valor `FALSO` se asocia al valor `0`; de no proceder de esta forma, las operaciones con objetos de tipo `boole` tendrán un funcionamiento erróneo.

Expresiones.

No todos los operadores que se comentaron en la sección 2.1 se pueden utilizar directamente en `C`. En este subapartado se pretende comentar sólo los operadores que el lenguaje permite manejar directamente, y especificar su sintaxis correcta.

Operadores aritméticos: Se englobará en este grupo los operadores enteros y reales. Es conveniente recordar el concepto de *polisemia*, ya que la mayor parte de los operadores que se describen son válidos para el tipo entero o el real y, dependiendo del tipo de los operandos, expresan la correspondiente operación entera o real. También es preciso recordar el concepto de *coerción* entre enteros y reales. Se comentará, de nuevo, más ampliamente al hablar de conversión de tipos. Los operadores aritméticos básicos del lenguaje `C` son:

- `+` : puede ser un operador *unario*, indicando el signo o *binario*, representando la suma.
- `-` : puede ser un operador *unario*, indicando el signo o *binario*, representando la resta.
- `*` es el operador binario que denota la multiplicación.
- `/` es el operador binario para la división.

  Si los operandos son enteros, denota el *cociente* de la división entera. Si al menos uno de ellos es real, denota la división real. Como ejemplo, notése que en el algoritmo de la media aritmética la última instrucción ha pasado a ser

  ```
  media = suma/4.0
  ```
- `%` es el operador que representa al *resto* de la división entera.

Esta lista de operadores básicos no incluye muchos de los operadores o funciones aritméticas que suelen utilizarse en el diseño de algoritmos, como la exponenciación, o la raíz cuadrada.

Esto no quiere decir que no se puedan utilizar: el lenguaje `C` es bastante reducido y, para completarlo, va acompañado de un conjunto de funciones conocido como *biblioteca estándar* y que le añade todas las funcionalidades de las que carece. En concreto, para la exponenciación y la raíz cuadrada dispone de las funciones ya definidas `pow(x,n)` y `sqrt(x)` en la biblioteca `math.h`. Cuando se quiere utilizar alguna de las funciones de la biblioteca estándar hay que indicar qué parte de ella se utilizará, con una directiva `#include` del preprocesador; en el caso del ejemplo anterior,

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

Operadores lógicos: `C` es un lenguaje que contiene implícitamente el tipo boole, es decir, no tiene el tipo de forma explícita, pero puede operar con valores lógicos, asociando el valor 0 a `falso` y cualquier valor distinto de 0 a `cierto`.

- Operadores relacionales: Como ya se dijo, los operadores relacionales se utilizan para comparar valores del mismo tipo (o compatible), obteniendo como resultado un valor lógico. En el lenguaje `C` son:
  - == es el operador igual.
  - != es el operador distinto.
  - < es el operador menor.
  - <= es el operador menor o igual.
  - > es el operador mayor.
  - >= es el operador mayor o igual.

  Es un error muy habitual utilizar "=" (que denota una asignación) en lugar de "==" (que es el operador relacional de igualdad) para ver si dos valores son o no iguales. En otros lenguajes esto provocaría un error sintáctico; sin embargo, en `C` el compilador no avisará del error y puede ser muy difícil de detectar.
- Conectores: Como también se comentó, unen operandos de tipo lógico dando como resultado un valor lógico. La notación del lenguaje `C` es:
  - `&&` es el operador `AND`. El resultado de aplicar este operador sobre dos operandos lógicos será `cierto` si ambos lo son, `falso` en otro caso.

- `||` es el operador `OR`. El resultado de aplicar este operador sobre dos operandos lógicos será `falso` si ambos lo son, `cierto` en otro caso.
- `!` es el operador `NOT`. Es un operador unario que invierte el valor de una expresión lógica.

Además, el lenguaje `C` presenta pecualiaridades propias, relacionadas con cómo evaluar las expresiones que forman parte de un programa. Cabe destacar las siguientes:

Precedencia y Asociatividad: Cuando en una expresión aparece más de un operador, deben existir unas reglas para saber en qué orden se aplican los distintos operadores para obtener el resultado final de la expresión. Dichas reglas vienen marcadas por la *precedencia* y la *asociatividad*.

La *precedencia* indica qué operadores se evalúan en primer lugar y cuáles después. En `C` se evalúan primero las expresiones que aparecen entre paréntesis; por lo tanto, en caso de duda, es aconsejable utilizarlos.

La *asociatividad* indica en qué orden se evalúa una expresión en la que aparecen varios operadores con la misma precedencia. Los operadores de `C` que se han presentado tienen mayoritariamente una asociatividad de izquierda a derecha; es decir, si dos operadores en una expresión tienen la misma precedencia, se evalúa la expresión de izquierda a derecha. Sin embargo, el cambio de signo y la negación (`!`) tienen asociatividad de derecha a izquierda.

La siguiente tabla muestra la precedencia de los operadores de `C` vistos, de mayor a menor (de arriba hacia abajo). También muestra la asociatividad de cada uno de estos operadores.

Operador	Asociatividad
!	de derecha a izquierda
− (cambio de signo)	de derecha a izquierda
* / %	de izquierda a derecha
+ −	de izquierda a derecha
< <= > >=	de izquierda a derecha
== !=	de izquierda a derecha
&&	de izquierda a derecha
||	de izquierda a derecha

Conversión de tipos: También se conoce como "*cast*" y consiste en convertir un objeto de un tipo de datos dado a otro tipo, para realizar una determinada operación. En muchos casos el `C` realiza automáticamente dicha conversión: de nuevo aparece el concepto de coerción que se traduce en, por ejemplo, permitir que se realice una división entre un real y un entero (`3.5/2`), sabiendo que el entero se transforma a su equivalente real (`2.0`) antes de hacer dicha división y que el resultado será real y sólo podrá asignarse a un objeto de tipo real.

Sin embargo, en ocasiones se precisa hacer la conversión explícitamente. Para ello, se indica entre paréntesis a qué tipo se quiere convertir un determinado objeto, que se escribe a continuación. Por ejemplo, `(float)3` convierte la constante entera `3` en la constante real `3.0` (aunque matemáticamente sea el mismo valor, la representación interna es distinta).

2.2.3. Entrada/Salida de Datos.

Al diseñar algoritmos, será normal asumir que se conocen los datos (y que se emitirán convenientemente los resultados). Pero cuando se realice el programa, se deberá disponer de algún tipo de herramienta de comunicación. Si se retoma el ejemplo de la media aritmética,

```
int main() {
        /* Datos */
        int num1, num2, num3, num4;
        /* Resultados */
        float media;

        int suma;

        suma = num1 + num2 ;
        suma = suma + num3 ;
        suma = suma + num4 ;
        media = suma / 4.0 ;
}
```

como algoritmo es completamente correcto; el compilador lo considerará válido, pero si se ejecutara no se realizaría ningún cálculo: el programa no puede realizarlos si no se instancian los valores de los datos, es decir, si no se dan valores concretos a `num1`, `num2`, `num3` y `num4` que permitan que las asignaciones sobre `suma` se puedan realizar correctamente.

Para poder leer datos de la entrada estándar (normalmente, el teclado), se utiliza una función de la biblioteca estándar, la función `scanf`. Si se introduce dicha función en el código de la media aritmética se producen varios cambios:

```
1  /* Ficheros de Cabecera */
2  #include <stdio.h>
3
4  int main() {
5          /* Datos */
6          int num1, num2, num3, num4;
7          /* Resultados */
8          float media;
9
10         int suma;
11
12         /* Lectura de datos */
13         scanf("%d", &num1);
14         scanf("%d", &num2);
15         scanf("%d", &num3);
16         scanf("%d", &num4);
17
18         suma = num1 + num2 ;
19         suma = suma + num3 ;
20         suma = suma + num4 ;
21         media = suma / 4.0 ;
22 }
```

- En la línea 2, se debe añadir la directiva `#include <stdio.h>`, que indica que se utilizarán funciones de entrada/salida (**st**an**d**ard **i**nput/**o**utput en inglés).
- En las líneas de la 13 a la 16, se está especificando la lectura de dichos valores.

La función `scanf` se utiliza para leer datos de la entrada estándar y almacenarlos en una variable[8]: se debe indicar entre comillas el tipo de datos que se quiere leer, mediante un código de control, y a continuación, separado por una coma, la variable (precedida por el símbolo &)[9] donde se quiere almacenar el dato leído.

Un *código de control* indica el tipo de dato que se va a tratar y está formado por el símbolo "%" seguido de un carácter: "d" para los enteros, "f" para los reales y "c" para los caracteres (para el tipo boole se podría utilizar también el carácter "d").

Otros ejemplos del uso de `scanf`, serían:

```
scanf("%c",&opcion);
scanf("%f",&Peso2);
```

La versión actual del cálculo de la media podría compilarse y ejecutarse. Pero sería un programa completamente inútil, ya que ahora sí que se realizan correctamente los cálculos, pero no ofrece ningún resultado a un posible usuario. Una posible solución sería mostrar por la salida estándar (que, normalmente, es el monitor) el valor que se asigna a la variable `media`. Para hacerlo, es preciso utilizar la función, también de la biblioteca estándar, `printf`. La nueva versión del programa de la media aritmética sería la siguiente:

```
1  /* Ficheros de Cabecera */
2  #include <stdio.h>
3
4  int main() {
5          /* Datos */
6          int num1, num2, num3, num4;
7          /* Resultados */
8          float media;
9
10         int suma;
11
12         /* Lectura de datos */
13         scanf("%d", &num1);
14         scanf("%d", &num2);
15         scanf("%d", &num3);
16         scanf("%d", &num4);
17
18         suma = num1 + num2 ;
19         suma = suma + num3 ;
20         suma = suma + num4 ;
21         media = suma / 4.0 ;
22
23
```

[8]Se trata de una función muy completa y compleja (y que, habitualmente, produce muchos dolores de cabeza); aquí se muestra en su forma más simple.

[9]Más adelante, cuando se estudien los punteros, se explicará el significado del símbolo &.

```
24         /* Escritura de resultados */
25         printf("%f", media);
26 }
```

La diferencia está en la línea 25, en la que se utiliza `printf`; tal y como se ha utilizado, muestra el valor de `media` en la salida estándar. Para ello, se ha vuelto a utilizar un código de control (`"%f"`, por `float`) y, a continuación y separado por comas, se indica qué objeto se visualiza.

Esta versión ya sería completamente correcta, se puede compilar, ejecutar y dispone de la capacidad de comunicarse con el exterior. Pero seguiría siendo un poco inútil desde el punto de vista de un usuario que no conociera el código (en caso de incredulidad, se recomienda encarecidamente que se edite el programa, se compile y ... se pida a un amigo que lo ejecute).

El usuario no tiene ninguna ayuda que le permita saber lo que ocurre: ningún mensaje de ayuda, nada que le permita deducir que se están esperando sus datos para producir un resultado.

Una versión más "políticamente correcta" del mismo programa sería la siguiente, que permite comentar más peculiaridades sobre el uso de `printf` y `scanf`:

```
1  /* Ficheros de Cabecera */
2  #include <stdio.h>
3
4  int main() {
5          /* Datos */
6          int num1, num2, num3, num4;
7          /* Resultados */
8          float media;
9
10         int suma;
11
12         printf("Programa para calculo de la media de 4 enteros:\n"):
13         /* Lectura de datos */
14         printf("\tPrimer numero:");
15         scanf("%d", &num1);
16         printf("\n\tSegundo numero:");
17         scanf("%d", &num2);
18         printf("\n\tTercer numero:");
19         scanf("%d", &num3);
20         printf("\n\tCuarto numero:");
21         scanf("%d", &num4);
22
23         suma = num1 + num2 ;
24         suma = suma + num3 ;
25         suma = suma + num4 ;
26         media = suma / 4.0 ;
27
28         /* Escritura de resultados */
29         printf("\n\nLa media aritmetica de %4d, %4d, %4d y %4d es %7.2f.", num1,
30         num2, num3, num4, media);
31 }
```

Lo que se ha hecho ha sido aprovechar la posibilidad que ofrece la función `printf` de mostrar mensajes, además de visualizar valores de variables. El efecto de `printf("mensaje")` es mos-

trar `mensaje` en la salida estándar. Es lo que se ha hecho en la líneas 12, 14, 16, 18 y 20. Si se quiere además visualizar el valor de un objeto o más objetos, como se ha hecho en la línea 29, en el cuerpo del mensaje hay que poner un código de control (por cada dato) en el punto del mensaje donde se quiere visualizar y, a continuación del mensaje y separado por comas, los objetos.

Además, en el código de control, entre el símbolo `%` y el carácter que indica el tipo, se puede poner un número (`%nd`) que indica que se han de utilizar `n` dígitos para visualizar el dato. Si se trata de un dato real, se puede poner un número decimal (`%n.mf`) que indica que se han de utilizar `n` dígitos para visualizar el dato y, de ellos, se han de mostrar `m` decimales. Hay que hacer notar que esto no modifica el valor de los objetos, sólo influye en la forma en que la función `printf()` los visualiza.

También se han utilizado otros códigos, que empiezan por el carácter "\", y que representan carácteres especiales. Dos de los más utilizados son "\n", para realizar un salto de línea, y "\t", para introducir un tabulador.

Cabe destacar cómo el `C` realiza una distinción clara entre la entrada y la salida. Así, cuando se quiera mostrar un mensaje previo a la lectura de datos hay que utilizar la función `printf` y, a continuación, mediante la función `scanf` se indica que se va a leer un dato. Por ejemplo, tal y como se hace en las líneas 14 y 15 para leer el valor de `num1`.

Del ejemplo anterior, se desprende un segundo esquema básico de programa; con respecto al esquema presentado en la subsección 2.2.1, se añade la posibilidad de incluir ficheros de cabecera (para incluir funciones de la librería estándar o de otras librerías) y una organización básica *lectura-cuerpo de instrucciones-escritura* que se procurará mantener siempre que sea posible:

```
1  /* Ficheros de Cabecera */
2
3  /* Función principal */
4  int main() {
5
6      /* Especificación de datos */
7      /* Especificación de resultados  */
8
9      /* Declaración de variables de la función main*/
10
11     /* Lectura de datos */
12
13     /* Cuerpo del algoritmo */
14
15     /* Escritura de resultados */
16
17 }
```

Las funciones de entrada/salida proporcionan un método para establecer comunicación entre un algoritmo (o mejor, entre un programa) y el mundo exterior. Este método no es único y, de hecho, existe otro más general que se presentará al formalizar el concepto de parámetro cuando se definan funciones y procedimientos. En cualquier caso, el objetivo principal de la asignatura, no debe olvidarse, es el diseño lógico de las acciones que forman el cuerpo del algoritmo, independientemente del modo en que se consiga que se establezca la comunicación.

2.3. Resumen y Consideraciones de Estilo.

En esta tema se han presentado las herramientas básicas de un programador, los objetos que puede manipular para realizar un determinado proceso. Cada objeto está definido por tres atributos, el nombre, el tipo y el valor. Los objetos pueden combinarse en expresiones, que son combinaciones de operandos y operadores del mismo tipo. Un objeto variable puede cambiar de valor mediante la asignación, que es la acción básica en un entorno de programación.

Además de estos conceptos comunes a cualquier lenguaje de programación, en este tema se ha presentado también el tratamiento que de ellos realiza el lenguaje `C`.

El objetivo de un buen programador es escribir algoritmos claros e inteligibles, ya que esto facilitará su corrección y mantenimiento. A continuación se citan algunas buenas prácticas que es aconsejable seguir para conseguir un buen estilo de programación.

- Utilizar nombres significativos para las variables. Si una variable va a almacenar el peso de una persona, es bastante conveniente denominarla `peso`, `Peso`, `peso_pers`, `peso1`, ... y no darle un nombre del tipo `arfgh`, `axd`, `xt123bf` ... Obviamente todos estos nombres son válidos, pero los del primer grupo reflejan claramente qué es lo que almacena la variable, mientras que los del segundo grupo no.

  En esta asignatura, además, se utilizará un convenio bastante difundido y que consiste en utilizar mayúsculas para las constantes y minúsculas para nombrar a las variables (y funciones y procedimientos, y definiciones de tipos, como se verá más adelante). En el caso de que el nombre de la variable sea compuesto, se utilizará mayúscula al principio de cada nueva palabra (siguiendo con el ejemplo, `pesoPersona`).

- Es conveniente utilizar comentarios a lo largo del programa que indiquen qué es lo que está haciendo. Tampoco conviene abusar y poner más líneas de comentarios que de código.

A medida que se avance en la asignatura, se presentarán más consejos de estilo.

2.4. Glosario.

carácter especial: cualquier carácter que en un determinado lenguaje tiene un significado especial y cuyo uso debe restringirse.

coerción: conversión explícita de tipos.

constante: objeto cuyo valor nunca cambia.

instanciar: dar un valor concreto a un objeto, normalmente un parámetro de entrada.

identificador: nombre que identifica un objeto en un lenguaje de programación o en pseudocódigo. Más adelante se verá que también se aplica sobre funciones y procedimientos.

operador unario: operador que actúa sobre un único operando.

operador binario: operador que actúa sobre dos operandos.

palabra reservada: en un lenguaje de programación, o en pseucódigo, palabra cuyo uso está restringido puesto que es significativa en la interpretación del código ya que indica una acción especial, un tipo, un identificador del lenguaje...

parámetro: objeto que representa las necesidades de comunicación de un algoritmo. Su número y uso queda definido por el problema que se intenta resolver mediante el algoritmo.

polisemia: propiedad que permite identificar a un mismo símbolo con distintos operadores, por lo que para interpretarlo correctamente se ha de tener en cuenta el contexto en el que aparece.

precisión: número de cifras significativas con que se puede almacenar un valor real en formato exponente-mantisa y que depende del computador o del lenguaje de programación.

rango: conjunto de valores situados entre un valor máximo y un valor mínimo que un determinado computador o lenguaje de programación permite representar.

variable: objeto cuyo valor cambia a medida que se desarrollan acciones en un algoritmo. Su número y uso depende del método utilizado para resolver un problema.

2.5. Bibliografía.

1. ”Introducción a la Programación (vol. I)”, Biondi & Clavel. Ed. Masson. 1991.
2. “El Lenguaje de Programación `C`. Diseño e Implementación de Programas”. Félix García, Jesús Carretero, Javier Fernández & Alejandro Calderón. Pearson Education, Madrid 2002.
3. “The C Programming Language”. Brian W. Kernigham & Dennis M. Ritchie. Prentice-Hall Inc. 1988.

2.6. Actividades y Problemas Propuestos.

1. Se dispone de dos variables, `x` e `y`. No importa de qué tipo son. Hay que escribir un algoritmo que permita que intercambien sus valores.
2. Sabiendo que `x`, `y` y `z` son variables del mismo tipo, con valores distintos dos a dos, analizar la siguientes secuencias de instrucciones e indicar en cuáles varía el resultado si se invierte el orden de ejecución. Por ejemplo, si varía o no el resultado si se ejecuta la secuencia

$$\begin{array}{c} x = y \\ z = x \end{array} \quad \text{o la inversa} \quad \begin{array}{c} z = x \\ x = y \end{array}$$

$$(a)\ \begin{array}{l} x = y \\ z = y \end{array} \qquad (b)\ \begin{array}{l} x = y \\ y = x \end{array} \qquad (c)\ \begin{array}{l} x = x + y \\ z = x + y \end{array} \qquad (d)\ \begin{array}{l} y = x + y \\ z = x + z \end{array}$$

3. Escribir un algoritmo que, dados tres valores de tipo real, permita calcular su media aritmética.
4. Diseñar un algoritmo que permita pasar una cantidad de tiempo expresada en segundos al formato horas, minutos y segundos.
5. Sabiendo que el precio de venta al público de un artículo engloba su precio de coste y las siguientes repercusiones,
   - Gastos de comercialización, estimados en un 20 % sobre el precio de coste,
   - Gastos Generales, personal, publicidad ... estimados en un 75 % sobre el precio de coste,

- Impuestos, que repercuten un 1.5 % sobre el precio de coste,
- Y al resultado del precio obtenido, añadirle un 0.7 % sobre el precio resultante como aportaciones a ONGs,

escribir un algoritmo que, conociendo el precio de coste, permita determinar el precio de venta al público final.

6. Escribir un algoritmo que permita determinar la solución de una ecuación de primer grado, `ax + b = c`.
7. Escribir un algoritmo que, dado un valor de tipo real, permita calcular el área del cuadrado cuyo lado tiene esa medida.
8. Escribir un algoritmo que, dado un valor de tipo real que representa el área de un círculo, permita calcular el perímetro de su circunferencia.