

Programación orientada a objetos *versus* programación estructurada: C++ y algoritmos

Introducción

El aprendizaje de la programación requiere el conocimiento de técnicas y metodologías de *programación estructurada*. Aunque a finales del siglo XX y, sobre todo en este siglo XXI, la *programación orientada a objetos* se ha convertido en la tecnología de software más utilizada; el conocimiento profundo de algoritmos y estructuras de datos, en muchos casos con el enfoque estructurado, facultará al lector y futuro programador los fundamentos técnicos necesarios para convertirse en un brillante programador de C++, en general, y programador orientado a objetos, en particular.

1.1. Concepto de algoritmo

Un **algoritmo** es una secuencia finita de instrucciones, reglas o pasos que describen de modo preciso las operaciones que una computadora debe realizar para ejecutar una tarea determinada en un tiempo finito [Knuth 68]¹. En la práctica, un algoritmo es un método para resolver problemas mediante los pasos o etapas siguientes:

1. *Diseño del algoritmo* que describe la secuencia ordenada de pasos —sin ambigüedades— conducentes a la solución de un problema dado (*Análisis del problema y desarrollo del algoritmo*).
2. Expresar el algoritmo como un *programa* en un lenguaje de programación adecuado. (*Fase de codificación*).
3. *Ejecución y validación* del programa por la computadora.

Para llegar a la realización de un programa es necesario el diseño previo de un algoritmo indicando *cómo* hace el algoritmo la tarea solicitada, y eso se traduce en la construcción de un algoritmo. El resultado final del diseño es una solución que debe ser fácil de traducir a estructuras de datos y estructuras de control de un lenguaje de programación específico.

Las dos herramientas más comúnmente utilizadas para diseñar algoritmos son: *diagramas de flujo* y *pseudocódigos*.

- **Diagrama de flujo (*flowchart*)**. Representación gráfica de un algoritmo.
- **Pseudocódigo**. *Lenguaje de especificación de algoritmos*, mediante palabras similares al inglés o español.

¹ Donald E. Knuth (1968): *The art of Computer Programming*, vol. 1, 1.ª ed., 1968; 2.ª ed. 1997, Addison Wesley. Knuth, es considerado uno de los padres de la algoritmia y de la programación. Su trilogía sobre “Programación de computadoras” es referencia obligada en todo el mundo docente e investigador de Informática y Computación.

El **algoritmo** es la especificación concisa del método para resolver un problema con indicación de las acciones a realizar. Un algoritmo es un conjunto finito de reglas que dan una secuencia de operaciones para resolver un determinado problema. Es, por tanto, *un método para resolver un problema que tiene en general una entrada y una salida*. Las características fundamentales que debe cumplir todo algoritmo son:

- Un algoritmo debe ser *preciso* e indicar el orden de realización de cada paso.
- Un algoritmo debe estar bien *definido*. Si se sigue un algoritmo dos veces, se debe obtener el mismo resultado cada vez.
- Un algoritmo debe ser *finito*. Si se sigue un algoritmo, se debe terminar en algún momento; o sea, debe tener un número finito de pasos.

La definición de un algoritmo debe describir tres partes: *Entrada, Proceso y Salida*.

EJEMPLO 1.1. *Se desea diseñar un algoritmo para conocer si un número es primo o no.*

Un número es primo si sólo puede dividirse por sí mismo y por la unidad (es decir, no tiene más divisores que él mismo y la unidad). Por ejemplo: 9, 8, 6, 4, 12, 16, 20, etc., no son primos, ya que son divisibles por números distintos a ellos mismos y a la unidad. Así, 9 es divisible por 3, 8 lo es por 2, etc. El algoritmo de resolución del problema pasa por dividir sucesivamente el número por 2, 3, 4..., etc.

Entrada: dato n entero positivo

Salida: es o no primo.

Proceso:

1. Inicio.
2. Poner x igual a 2 ($x = 2$, x variable que representa a los divisores del número que se busca n).
3. Dividir n por x (n/x).
4. Si el resultado de n/x es entero, entonces n es un número primo y bifurcar al punto 7; en caso contrario, continuar el proceso.
5. Suma 1 a x ($x \leftarrow x + 1$).
6. Si x es igual a n, entonces n es un número primo; en caso contrario, bifurcar al punto 3.
7. Fin.

El algoritmo anterior escrito en pseudocódigo es:

```
algoritmo primo
1. inicio
   variables
     entero: n, x:
     lógico: primo;
2. leer(n);
   x←2;
   primo←verdadero;
3. mientras primo y (x < n) hacer
4.   si n mod x != 0 entonces
5.     x← x+1
     sino
       primo ←falso
   fin si
fin mientras
si (primo) entonces
  escribe('es primo')
sino
  escribe('no es primo')
fin si
7. fin
```

1.2. Programación estructurada

La programación estructurada consiste en escribir un programa de acuerdo con unas reglas y un conjunto de técnicas. Las reglas son: el programa tiene un diseño modular, los módulos son diseñados descendientemente, cada módulo de programa se codifica usando tres estructuras de control (*secuencia, selección e iteración*); es el conjunto de técnicas que han de incorporar: recursos abstractos; diseño descendente y estructuras básicas de control.

Descomponer un programa en términos de *recursos abstractos* consiste en descomponer acciones complejas en términos de acciones más simples capaces de ser ejecutadas en una computadora.

El diseño *descendente* se encarga de resolver un problema realizando una descomposición en otros más sencillos mediante módulos jerárquicos. El resultado de esta jerarquía de módulos es que cada módulo se refina por los de nivel más bajo que resuelven problemas más pequeños y contienen más detalles sobre los mismos.

Las *estructuras básicas de control* sirven para especificar el orden en que se ejecutarán las distintas instrucciones de un algoritmo. Este orden de ejecución determina el *flujo de control* del programa.

La programación estructurada significa:

- El programa completo tiene un diseño modular.
- Los módulos se diseñan con metodología descendente (puede hacerse también ascendente).
- Cada módulo se codifica utilizando las tres estructuras de control básicas: secuenciales, selectivas y repetitivas (ausencia total de sentencias **ir** → **a** (*goto*)).
- *Estructuración y modularidad* son conceptos complementarios (se solapan).

EJEMPLO 1.2. *Calcular la media de una serie de números positivos, suponiendo que los datos se leen desde un terminal. Un valor de cero —como entrada— indicará que se ha alcanzado el final de la serie de números positivos.*

El primer paso a dar en el desarrollo del algoritmo es descomponer el problema en una serie de pasos secuenciales. Para calcular una media se necesita sumar y contar los valores. Por consiguiente, el algoritmo en forma descriptiva sería:

```

inicio
  1. Inicializar contador de números C y variable suma S a cero (S←0, C←1).
  2. Leer un número en la variable N (leer(N))
  3. Si el número leído es cero: (si (N =0) entonces)
    3.1. Si se ha leído algún número (Si C>0)
      • calcular la media; (media← S/C)
      • imprimir la media; (Escribe(media))
    3.2. si no se ha leído ningún número (Si C=0))
      • escribir no hay datos.
    3.3. fin del proceso.
  4. Si el numero leído no es cero : (Si (N <> 0) entonces)
      • calcular la suma; (S← S+N)
      • incrementar en uno el contador de números; (C←C+1)
      • ir al paso 2.

```

Fin

algoritmo escrito en pseudocódigo:

```

algoritmo media
inicio
variables
entero: n, c, s;
real: media;
  C← 0;
  S←0;

```

```
repetir
  leer(N)
  Si N <> 0 Entonces
    S←S+N;
    C←C+1;
  fin si
hasta N=0
si C>0 entonces
  media ← S/C
  escribe(media)
sino
  escribe('no datos')
fin si
fin
```

Un programa en un lenguaje procedimental es un conjunto de instrucciones o sentencias. Lenguajes de programación como C, Pascal, FORTRAN, y otros similares, se conocen como *lenguajes procedimentales* (por procedimientos). Es decir, cada sentencia o instrucción indica al compilador que realice alguna tarea: obtener una entrada, producir una salida, sumar tres números, dividir por cinco, etc. En el caso de pequeños programas, estos principios de organización (denominados *paradigma*) se demuestran eficientes. El programador sólo ha de crear esta lista de instrucciones en un lenguaje de programación, compilar en la computadora y ésta, a su vez, ejecuta las instrucciones.

Cuando los programas se vuelven más grandes, la lista de instrucciones aumenta considerablemente, de modo tal que el programador tiene muchas dificultades para controlar ese gran número de instrucciones. Para resolver este problema los programas se descomponen en unidades más pequeñas que adoptan el nombre de *funciones* (*procedimientos*, *subprogramas* o *subrutinas* en otros lenguajes de programación). De este modo un programa orientado a procedimientos se divide en funciones, cada una de las cuales tiene un propósito bien definido y resuelve una tarea concreta, y se diseña una interfaz claramente definida (el prototipo o cabecera de la función) para su comunicación con otras funciones.

Con el paso de los años, la idea de romper un programa en funciones fue evolucionando y se llegó al agrupamiento de las funciones en otras unidades más grandes llamadas *módulos* (normalmente, en el caso de C++, denominadas **archivos** o **ficheros**); sin embargo, el principio seguía siendo el mismo: agrupar componentes que ejecutan listas de instrucciones (sentencias). Existen varias razones de la debilidad de los programas estructurados para resolver problemas complejos. Tal vez las dos razones más evidentes son éstas. Primera, las funciones tienen acceso ilimitado a los datos globales; segundo, las funciones inconexas y datos, fundamentos del paradigma procedimental proporcionan un modelo pobre del mundo real. La **programación orientada a objetos** se desarrolló para tratar de paliar diversas limitaciones que se encontraban en anteriores enfoques de programación.

1.3. El paradigma de orientación a objetos

La programación orientada a objetos aporta un nuevo enfoque a los retos que se plantean en la programación estructurada cuando los problemas a resolver son complejos. Al contrario que la programación procedimental que enfatiza en los algoritmos, la POO enfatiza en los datos. En lugar de intentar ajustar un problema al enfoque procedimental de un lenguaje, POO intenta ajustar el lenguaje al problema. La idea es diseñar formatos de datos que se correspondan con las características esenciales de un problema. Los lenguajes orientados combinan en una única unidad o módulo, tanto los datos como las funciones que operan sobre esos datos. Tal unidad se llama **objeto**. Si se desea modificar los datos de un objeto, hay que realizarlo mediante las funciones miembro del objeto. Ninguna otra función puede acceder a los datos. Esto simplifica la escritura, depuración y mantenimiento del programa.

En el paradigma orientado a objetos, el programa se organiza como un conjunto finito de objetos que contienen datos y operaciones (*funciones miembro en C++*) que llaman a esos datos y que se comunican entre sí mediante *mensajes*.

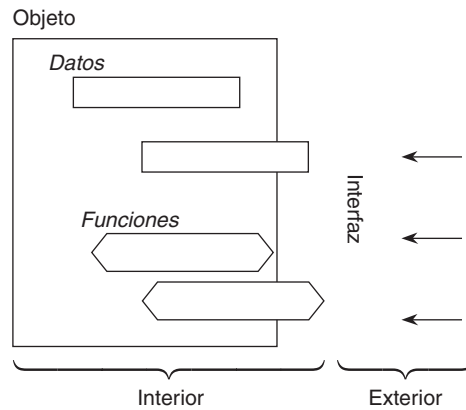


Figura 1.1. Diagrama de un objeto

Cuando se trata de resolver un problema con orientación a objetos, dicho problema no se descompone en funciones como en programación estructurada tradicional, caso de C, sino en objetos. El pensar en términos de objetos tiene una gran ventaja: se asocian los objetos del problema a los objetos del mundo real.

1.3.1. PROPIEDADES FUNDAMENTALES DE LA ORIENTACIÓN A OBJETOS

Existen diversas características ligadas a la orientación a objetos. Todas las propiedades que se suelen considerar, no son exclusivas de este paradigma, ya que pueden existir en otros paradigmas, pero en su conjunto definen claramente los lenguajes orientados a objetos. Estas propiedades son:

- **Abstracción (tipos abstractos de datos y clases).**
- **Encapsulado de datos.**
- **Ocultación de datos.**
- **Herencia.**
- **Polimorfismo.**

C++ soporta todas las características anteriores que definen la orientación a objetos.

1.3.2. ABSTRACCIÓN

La **abstracción** es la propiedad de los objetos que consiste en tener en cuenta sólo los aspectos más importantes desde un punto de vista determinado y no tener en cuenta los restantes aspectos. Durante el proceso de abstracción es cuando se decide qué características y comportamiento debe tener el modelo. Un medio de reducir la complejidad es la abstracción. Las características y los procesos se reducen a las propiedades esenciales, son resumidas o combinadas entre sí. De este modo, las características complejas se hacen más manejables.

En estructuras o registros, las propiedades individuales de los objetos se pueden almacenar en los miembros. Para los objetos es de interés no sólo *cómo* están organizados sino también *qué* se puede hacer con ellos; es decir, las operaciones de un objeto son también importantes. El primer concepto en el mundo de la orientación a objetos nació con los tipos abstractos de datos (**TAD**). Un **tipo abstracto de datos** describe no sólo los atributos de un objeto, sino también su comportamiento (las operaciones). Esto puede incluir también una descripción de los estados que puede alcanzar un objeto.

EJEMPLO 1.3. *Diferentes modelos de abstracción del término coche (carro).*

- Un coche (carro) es la combinación (o composición) de diferentes partes, tales como motor, carrocería, cuatro ruedas, cinco puertas, etc.
- Un coche (carro) es un concepto común para diferentes tipos de coches. Pueden clasificarse, por el nombre del fabricante (Audi, BMW, SEAT, Toyota, Chrysler...), por su categoría (turismo, deportivo, todoterreno...), por el carburante que utilizan (gasolina, gasoil, gas, híbrido...).

La abstracción coche se utilizará siempre que la marca, la categoría o el carburante no sean significativos. Así, un carro (coche) se utilizará para transportar personas o ir de Carchelejo a Cazorla, o de Paradinás a Mazarambrós.

1.3.3. ENCAPSULACIÓN Y OCULTACIÓN DE DATOS

El *encapsulado* o *encapsulación de datos* es el proceso de agrupar datos y operaciones relacionadas bajo la misma unidad de programación. En el caso de que los objetos posean las mismas características y comportamiento se agrupan en clases (unidades o módulos de programación que encapsulan datos y operaciones).

La ocultación de datos permite separar el aspecto de un componente, definido por su *interfaz* con el exterior, de sus detalles internos de implementación. Los términos ocultación de la información (*information hiding*) y encapsulación de datos (*data encapsulation*) se suelen utilizar como sinónimos, pero no siempre es así y muy al contrario son términos similares pero distintos. En C++ no es lo mismo, ya que los datos internos están protegidos del exterior y no se pueden acceder a ellos más que desde su propio interior y, por tanto, no están ocultos. El acceso al objeto está restringido sólo a través de una interfaz bien definida.

1.3.4. GENERALIZACIÓN Y ESPECIALIZACIÓN

La *generalización* es la propiedad que permite compartir información entre dos entidades evitando la redundancia. En el comportamiento de objetos existen con frecuencia propiedades que son comunes en diferentes objetos y esta propiedad se denomina generalización.

Por ejemplo, máquinas lavadoras, frigoríficos, hornos de microondas, tostadoras, lavavajillas, etc., son todos electrodomésticos (aparatos del hogar). En el mundo de la orientación a objetos, cada uno de estos aparatos es una **subclase** de la clase Electrodoméstico y a su vez Electrodoméstico es una **superclase** de todas las otras clases (máquinas lavadoras, frigoríficos, hornos de microondas, tostadoras, lavavajillas...). El proceso inverso de la generalización por el cual se definen nuevas clases a partir de otras ya existentes se denomina *especialización*.

En orientación a objetos, el mecanismo que implementa la propiedad de generalización se denomina **herencia**. La herencia permite definir nuevas clases a partir de otras clases ya existentes, de modo que presentan las mismas características y comportamiento de éstas, así como otras adicionales.

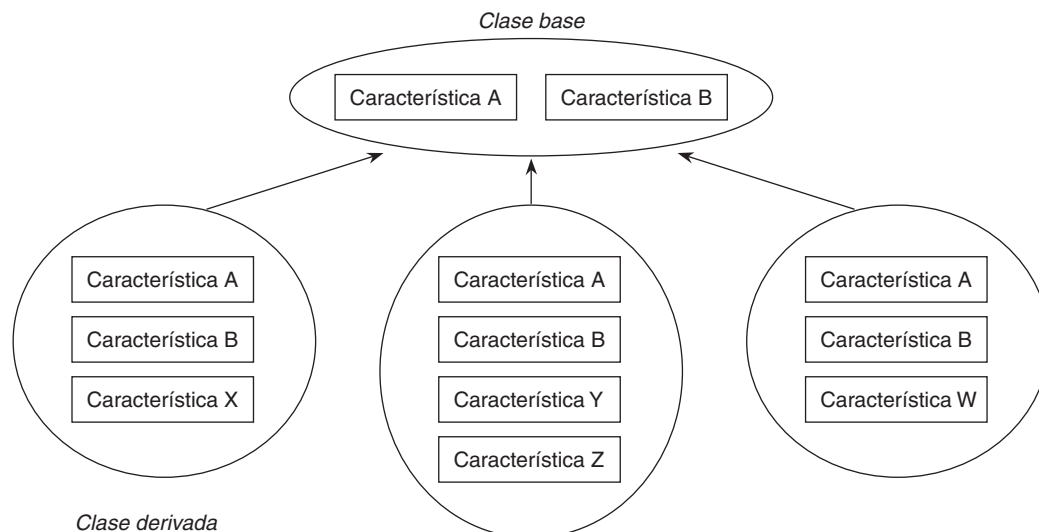


Figura 1.2. Jerarquía de clases.

Una clase *hereda* sus características (datos y funciones) de otra clase.

1.3.5. POLIMORFISMO

La propiedad de **polimorfismo** es aquella en que una operación tiene el mismo nombre en diferentes clases, pero se ejecuta de diferentes formas en cada clase. Así, por ejemplo, la operación *abrir* se puede dar en diferentes clases: abrir una puerta, abrir una ventana, abrir un periódico, abrir un archivo, abrir una cuenta corriente en un banco, abrir un libro, etc. En cada caso se ejecuta una operación diferente aunque tiene el mismo nombre en todos ellos "abrir". El polimorfismo es la propiedad de una operación de ser interpretada sólo por el objeto al que pertenece.

1.4. C++ Como lenguaje de programación orientada a objetos

C++ es una extensión (ampliación) de C con características más potentes. Estrictamente hablando es un superconjunto de C. Los elementos más importantes añadidos a C por C++ son: clases, objetos y programación orientada a objetos (C++ fue llamado originalmente “C con clases”).

En C++, un **objeto** es un elemento individual con su propia identidad; por ejemplo, un libro, un automóvil... Una **clase** puede describir las propiedades genéricas de un ejecutivo de una empresa (nombre, título, salario, cargo...) mientras que un objeto representará a un ejecutivo específico (Luis Mackoy, Lucas Soblechero).

En general, una clase define qué datos se utilizan para representar un objeto y las operaciones que se pueden ejecutar sobre esos datos. En el sentido estricto de programación, una clase es un tipo de datos. Diferentes variables se pueden crear de este tipo. En programación orientada a objetos, estas variables se llaman *instancias*. Las instancias son, por consiguiente, la realización de los objetos descritos en una clase. Estas instancias constan de datos o atributos descritos en la clase y se pueden manipular con las operaciones definidas dentro de ellas.

Una **clase** es una descripción general de un conjunto de objetos similares. Por definición todos los objetos de una clase comparten los mismos atributos (*datos*) y las mismas operaciones (*métodos*). Una clase encapsula las abstracciones de datos y operaciones necesarias para describir una entidad u objeto del mundo real.

El diseño de clases fiables y útiles puede ser una tarea difícil. Afortunadamente los lenguajes de programación orientada a objetos facilitan la tarea ya que incorporan clases existentes en su propia programación. Uno de los beneficios reales de C++ es que permite la reutilización y adaptación de códigos existentes y ya bien probados y depurados.

En el diseño de programas orientados a objetos se realiza en primer lugar el diseño de las clases que representan con precisión aquellas cosas que trata el programa; por ejemplo, un programa de dibujo, puede definir clases que representan rectángulos, líneas, pinceles, colores, etc. Las definiciones de clases, incluyen una descripción de operaciones permisibles para cada clase, tales como desplazamiento de un círculo o rotación de una línea. A continuación se prosigue el diseño de un programa utilizando objetos de las clases.

Los términos *objeto* e *instancia* se utilizan frecuentemente como sinónimos (especialmente en C++). Si una variable de tipo `Carro` se declara, se crea un objeto `Carro` (una instancia de la clase `Carro`). Las operaciones definidas en los objetos se llaman *métodos*. Cada operación llamada por un objeto se interpreta como un *mensaje* al objeto, que utiliza un método específico para procesar la operación.

Desde el punto de vista de implementación un **objeto** es una entidad que posee un conjunto de *datos* y un conjunto de *operaciones* (*funciones* o *métodos*). El *estado* de un objeto viene determinado por los valores que toman sus datos, cuyos valores pueden tener las restricciones impuestas en la definición del problema. Los **datos** se denominan también *atributos* y componen la estructura del objeto y las **operaciones** —también llamadas *métodos*— representan los servicios que proporciona el objeto.

Nuestro objetivo es ayudarle a escribir programas orientado a objetos tan pronto como sea posible. Sin embargo, como ya se ha observado, muchas características de C++ se han heredado de C, de modo que, aunque la estructura global de un programa pueda ser orientado objetos, consideramos que usted necesita conocimientos profundos del “viejo estilo *procedimental*”. Por ello, los capítulos de la primera parte del libro le van introduciendo lenta y pausadamente en las potentes propiedades orientadas a objetos de las últimas partes, al objeto de conseguir a la terminación del libro el dominio de la programación en C++.

NOTA: Compiladores y compilación de programas en C++

Existen numerosos compiladores de C++ en el mercado. Desde ediciones gratuitas y *descargables* a través de Internet hasta profesionales, con costes diferentes, comercializados por diferentes fabricantes. Es difícil dar una recomendación al lector porque casi todos ellos son buenos compiladores, muchos de ellos con Entornos Integrados de Desarrollo (**EID**). Si usted es estudiante, tal vez la mejor decisión sea utilizar el compilador que le haya propuesto su profesor y que utilice en su Universidad, Instituto Tecnológico o cualquier otro Centro de Formación, donde estudie. Si usted es un lector autodidacta y está aprendiendo por su cuenta existen varias versiones gratuitas que puede descargar desde Internet. Algunos de los más reconocidos son: **Dev-C++** de **Bloodshed** que cumple fielmente el estándar ANSI/ISO C++ (utilizado por los autores del libro para editar y compilar todos los programas incluidos en el mismo) y que corre bajo entornos Windows; **GCC** de GNU que corre bajo los entornos Linux y Unix. Existen muchos otros compiladores gratuitos por lo que tiene donde elegir. Tal vez un consejo más: procure que sea compatible con el estándar ANSI/ISO C++.

Bjarne Stroustrup (creador e inventor de C++) en su página oficial² ha publicado el 9 de febrero de 2006, “una lista in-

² <http://public.research.att.com/~bs/compilers.html>. El artículo “An incomplete list of C++ compilers” lo suele modificar Stroustrup y en la cabecera indica la fecha de modificación. En nuestro caso, consultamos dicha página mientras escribíamos el prólogo en la primera quincena de Marzo, y la fecha incluida es la de 9 de febrero de 2006.

completa de compiladores de C++”, que le recomiendo lea y visite. Por su interés incluimos, a continuación, un breve extracto de su lista recomendada:

Compiladores gratuitos

Apple C++
 Borland C++
 Dev-C++ de Bloodshed
 GNUIntel C++ para Linux
 Microsoft Visual C++ Toolkit 2003
 Sun Studio...

Compiladores comerciales

Borland C++
 Compaq C++
 HP C++
 IBM C++
 Intel C++ para Windows, Linux y algunos sistemas empo-
 trados
 Microsoft C++

Stroustrup recomienda un sitio de compiladores gratuitos de C y C++ (Compilers.net).

EJERCICIOS

- 1.1. Escribir un algoritmo que lea tres números y si el primero es positivo calcule el producto de los tres números, y en otro caso calcule la suma.
- 1.2. Escribir un algoritmo que lea el radio de un círculo, y calcule su perímetro y su área.
- 1.3. Escribir un algoritmo que lea cuatro números enteros del teclado y calcule su suma.
- 1.4. Escribir un algoritmo que lea un número entero positivo n , y sume los n primeros número naturales.
- 1.5. Escribir un algoritmo que lea un número $n > 0$ del teclado y sume la serie siguiente:

$$\sum_{i=1}^n a_i \quad \text{si} \quad a_i = \begin{cases} i*i & \text{si } i \text{ es impar} \\ 2 & \text{si } i \text{ es par} \end{cases} \quad \text{y} \quad n=5$$

- 1.6. Definir una jerarquía de clases para: animal, insecto, mamíferos, pájaros, persona hombre y mujer. Realizar un definición en pseudocódigo de las clases.

SOLUCIÓN DE LOS EJERCICIOS

- 1.1. Se usan tres variables enteras Numero1, Numero2, Numero3, en las que se leen los datos, y otras dos variables Producto y Suma en las que se calcula o bien el producto o bien la suma.

Entrada Numero1, Numero2 y Numero 3
 Salida la suma o el producto

```

inicio
1 leer los tres número Numero1, Numero2, Numero3
2 si el Numero1 es positivo
  calcular el producto de los tres numeros
  escribir el producto
3 si el Numero1 es no positivo
  calcular la suma de los tres número
  escribir la suma
fin
```

El algoritmo en pseudocódigo es:

```

algoritmo Producto_Suma
variables
```

```

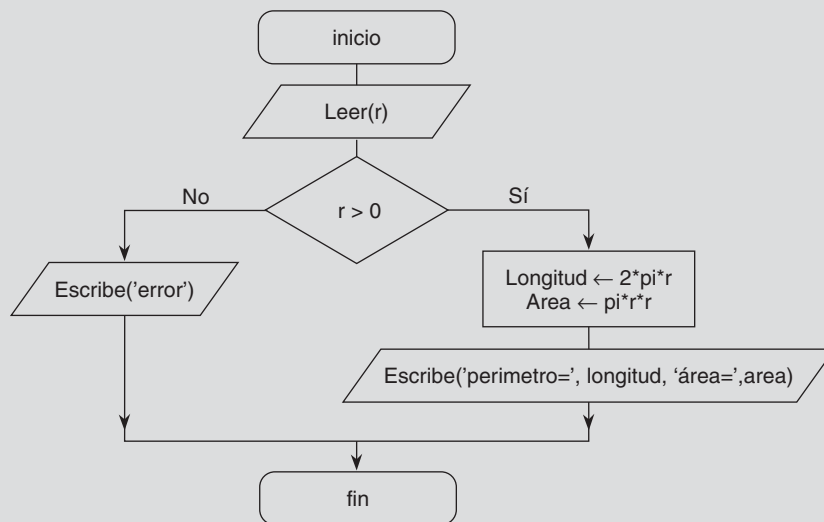
entero: Numero1, Numero2, Numero3, Producto, Suma
inicio
  Leer(Numero1, Numero2, Numero3)
  si (Numero1 > 0) entonces
    Producto ← Numero1* Numero2 * Numero3
    Escribe('El producto de los números es', Producto)
  sino
    Suma ← Numero1+ Numero2 + Numero3
    Escribe('La suma de los números es', Suma)
  fin si
fin
    
```

1.2. Se declaran las variables reales *r*, longitud y área, así como la constante *pi*

```

constantes pi = 3.14
variables
  real r, longitud, área
    
```

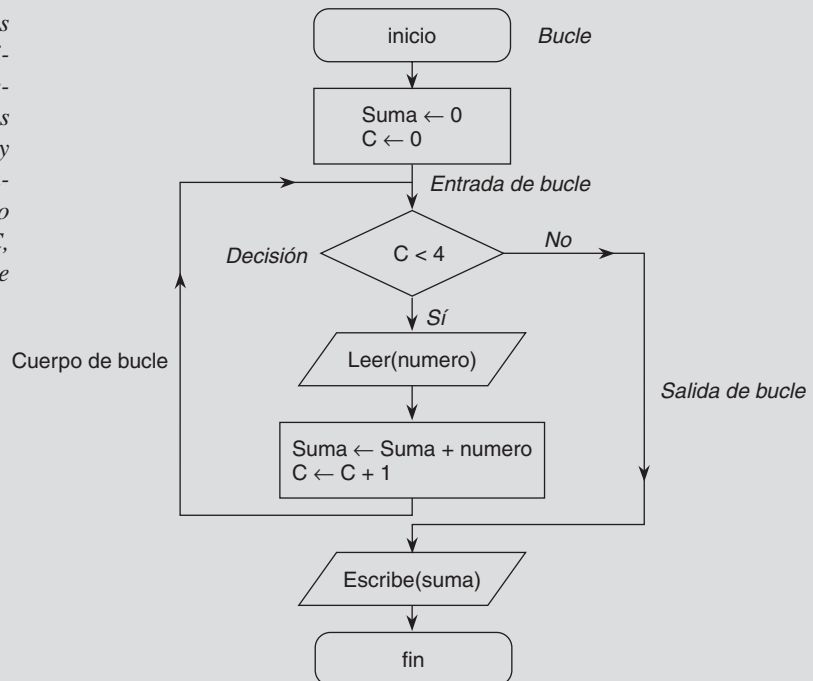
Diagrama de flujo



1.3. Un bucle es un segmento de programa cuyas instrucciones se repiten un número determinado de veces hasta que se cumple una determinada condición. Tiene las siguientes partes: entrada, salida, cuerpo del bucle y decisión que forma parte del cuerpo del bucle o bien de la entrada salida. El algoritmo se diseña con un bucle, un contador entero *C*, un acumulador entero *Suma* y una variable *numero* para leer los datos.

```

variables
  entero Suma, C, numero
    
```



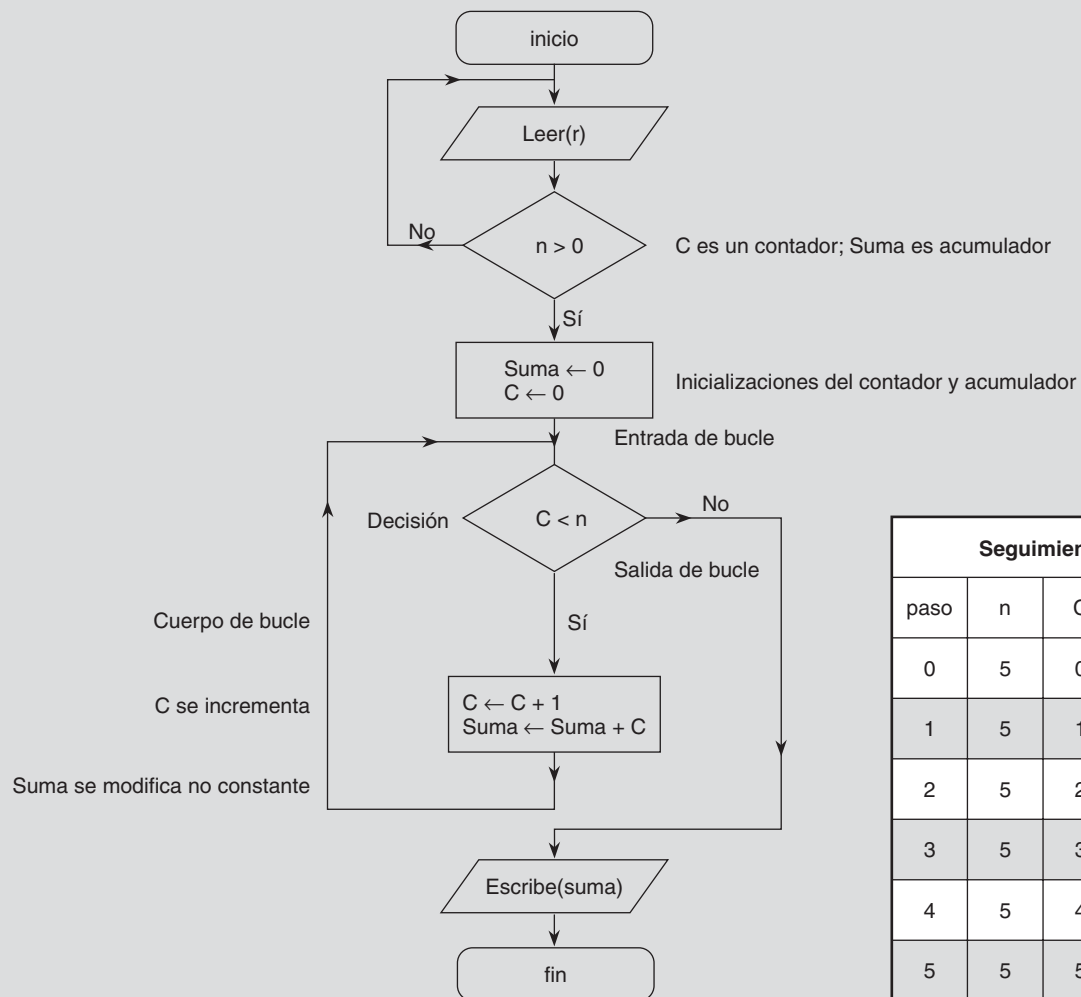
```

algoritmo suma_4
inicio
  variables
  entero: Suma, C, numero;
  C ← 0;
  suma ← 0;
  mientras C < 4 hacer
    leer(Numero)
    Suma ← Suma + numero;
    C ← C + 1;
  fin mientras
  escribe(suma)
fin

```

- 1.4. Inicialmente se asegura la lectura del número natural n positivo. Mediante un contador C se cuentan los números naturales que se suman, y en el acumulador $Suma$ se van obteniendo las sumas parciales. Además del diagrama de flujo se realiza un seguimiento para el caso de la entrada $n = 5$

variables Entero: n , $Suma$, C



```

algoritmo suma_n_naturales
inicio
variables
entero: Suma, C, n;
repetir
Leer(n)
hasta n>0
C ← 0;
suma ← 0;
mientras C < n Hacer
C ← C + 1;
suma ← Suma + C;
fin mientras
escribe(Suma)
fin

```

- 1.5. La estructura del algoritmo es muy parecida a la del ejercicio anterior, salvo que ahora en lugar de sumar el valor de una constante se suma el valor de una variable. Se usa un interruptor *sw* que puede tomar los valores verdadero o falso y que se encarga de decidir si la suma es el valor de $i*i$ o el valor de 2. Observar que en este ejercicio el contador *i* se inicializa a 1 por lo que el incremento de *i* se realiza al final del cuerpo del bucle, y la salida del bucle se realiza cuando $i > n$.

Pseudocódigo

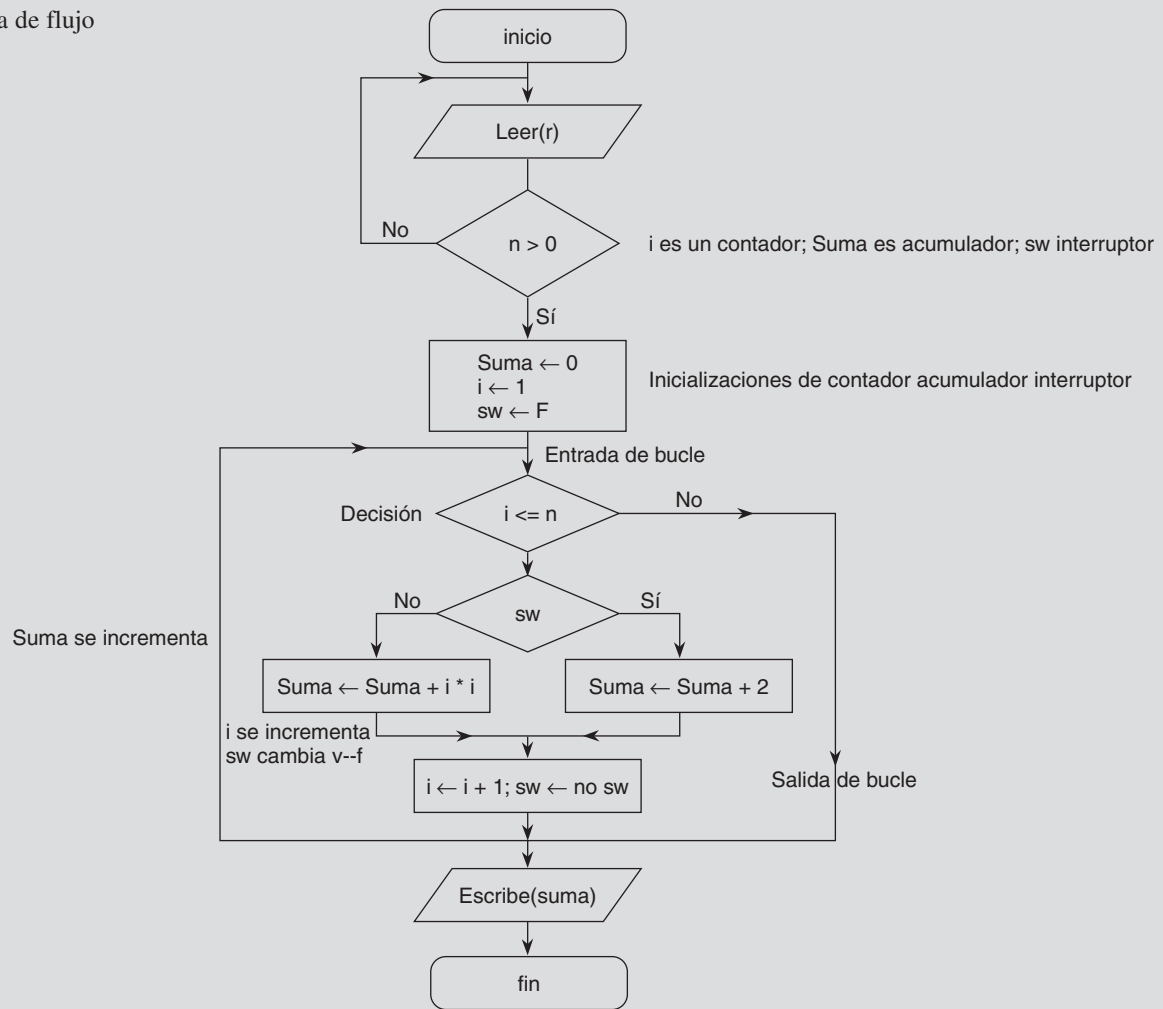
```

algoritmo suma_serie
inicio
variables
entero: Suma, i, n;
logico sw;
repetir
Leer(n);
hasta n>0
i ← 1;
suma ← 0;
Sw ← falso
mientras i <= n Hacer
si (sw) Entonces
Suma ← Suma + 2;
sino
Suma ← Suma + i*i
fin si
i ← i+1
sw ← no sw
fin mientras
escribe(Suma)
fin

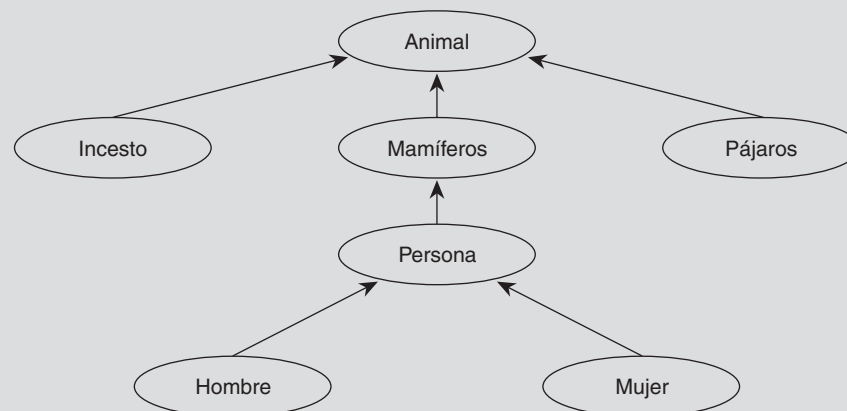
```

Seguimiento n=5				
paso	n	sw	i	suma
0	5	F	1	0
1	5	V	2	1
2	5	F	3	3
3	5	V	4	13
4	5	F	5	15
5	5	V	6	40

Diagrama de flujo



- 1.6. Las clases de objeto mamífero, pájaro e insecto se definen como subclases de animal; la clase de objeto persona, como una subclase de mamífero, y un hombre y una mujer son subclases de persona.



Las definiciones de clases para esta jerarquía puede tomar la siguiente estructura:

```
class criatura
  atributos (propiedades)
    string: tipo;
    real: peso;
    (...algun tipo de habitat...):habitat;
  operaciones
    crear()→ criatura;
    predadores(criatura)→ fijar(criatura);
    esperanza_vida(criatura) → entero;
    ...
fin criatura.

class mamifero hereda criatura;
  atributos (propiedades)
    real: periodo_gestacion;
  operaciones
    ...
fin mamifero.

class persona hereda mamifero;
  atributos (propiedades)
    string: apellidos, nombre;
    date: fecha_nacimiento;
    pais: origen;
fin persona.

class hombre hereda persona;
  atributos (propiedades)
    mujer: esposa;
    ...
  operaciones
    ...
fin hombre.

class mujer hereda persona;
  propiedades
    esposo: hombre;
    string: nombre;
    ...
fin mujer.
```

EJERCICIOS PROPUESTOS

- 1.1. Diseñar un algoritmo que lea e imprima una serie de números distintos de cero. El algoritmo debe terminar con un valor cero que no se debe imprimir. Visualizar el número de valores leídos.
- 1.2. Diseñar un algoritmo que imprima y sume la serie de números 4, 8, 12, 16,..., 400.
- 1.3. Escribir un algoritmo que lea cuatro números y a continuación imprima el mayor de los cuatro.
- 1.4. Diseñar un algoritmo para calcular la velocidad (en m/s) de los corredores de la carrera de 1.500 metros. La entrada (tiempo del corredor) consistirá en parejas de números (minutos, segundos); por cada corredor, el algo-

ritmo debe imprimir el tiempo en minutos y segundos así como la velocidad media. Ejemplo de entrada de datos: (3,53) (3,40) (3,46) (3,52) (4,0) (0,0); el último par de datos se utilizará como fin de entrada de datos.

- 1.5. Escribir un algoritmo que encuentre el salario semanal de un trabajador, dada la tarifa horaria y el número de horas trabajadas diariamente.
- 1.6. Escribir un algoritmo que cuente el número de ocurrencias de cada letra en una palabra leída como entrada. Por ejemplo, "Mortimer" contiene dos "m", una "o", dos "r", una "y", una "t" y una "e".

1.7. Dibujar un diagrama jerárquico de objetos que represente la estructura de un coche (carro).

1.8. Dibujar diagramas de objetos que representen la jerarquía de objetos del modelo *Figura geométrica*.

1.9. Escribir un algoritmo para determinar el máximo común divisor de dos números enteros (MCD) por el algoritmo de Euclides:

- Dividir el mayor de los dos enteros por el más pequeño.
- A continuación, dividir el divisor por el resto.
- Continuar el proceso de dividir el último divisor por el último resto hasta que la división sea exacta.
- El último divisor es el MCD.

NOTA PRÁCTICA SOBRE COMPILACIÓN DE PROGRAMAS FUENTE CON EL COMPILADOR DEV C++ Y OTROS COMPILADORES COMPATIBLES ANSI/ISO C++

Todos los programas del libro han sido compilados y ejecutados en el compilador de *freeware* (de libre distribución) BloodShed DEV-C++³, versión 4.9.9.2. Dev C++ es un editor de múltiples ventanas integrado con un compilador de fácil uso que permite la compilación, enlace y ejecución de programas C o C++.

Las sentencias `system("PAUSE");` y `return EXIT_SUCCESS;` se incluyen por defecto por el entorno **Dev** en todos los programas escritos en C++. El libro incluye ambas sentencias en prácticamente todos los códigos de programa. Por comodidad para el lector que compile los programas se han dejado estas sentencias. Sin embargo, si usted no utiliza el compilador Dev C++ o no desea que en sus listados aparezcan estas sentencias, puede sustituirlas bien por otras equivalentes o bien quitar las dos y sustituirlas por `return 0;` como recomienda Stroustrup para terminar sus programas.

`system("PAUSE");` detiene la ejecución del programa hasta que se pulsa una tecla. Las dos sentencias siguientes de C++ producen el mismo efecto:

```
cout << "Presione ENTER para terminar";
cint.get();
```

`return EXIT_SUCCESS;` devuelve el estado de terminación correcta del programa al sistema operativo. La siguiente sentencia de C++ estándar, produce el mismo efecto:

```
return 0;
```

Tabla 1.1. Dev C++ versus otros compiladores de C++ (en compilación)

Sustitución de sentencias	
DEV C++	Otros compiladores de C++
<code>system("PAUSE");</code>	<code>cout << "Presione ENTER para terminar"; cint.get();</code>
<code>return EXIT_SUCCESS;</code>	<code>return 0;</code>

³ <http://www.bloodshed.net/>