Capítulo 3

Programación Estructurada

Índice General

"Doing abominations is against the law, particularly if *the abominations are done* while *wearing a lobster bib."*
Woody Allen. "Without Feathers".

3.1. Introducción a la Programación Estructurada.

Cuando se definió en el tema 1 el concepto de algoritmo, se remarcó la palabra secuencia atendiendo a su doble significado: no se debe entender sólo que un algoritmo está formado por una relación más o menos larga de acciones, *sino que el orden en el que estas acciones aparecen es significativo*.

Esta consideración permite introducir el concepto de *estructura de control de flujo*. La *secuencia* es la más simple y consiste en realizar las acciones en el orden en que aparecen. Pero, además, a lo largo del algoritmo es posible encontrar que, dependiendo de ciertas circunstancias (de ciertas *condiciones*), se deben o no realizar ciertas acciones. Sería equivalente a realizar razonamientos del tipo

> "*si al resolver una ecuación de segundo orden el discriminante es negativo, se debe cambiar su signo y obtener dos raíces complejas conjugadas; si no, se obtienen dos raíces reales*".

Esto es lo que se conoce como una *estructura de control condicional*. Además, es posible que determinadas instrucciones haya que realizarlas más de una vez. Esta circunstancia se correspondería con razonamientos del tipo

> "*para obtener la expresión de un número en base dos, hay que dividirlo entre 2 hasta que el cociente sea igual a 1*".

En estos casos se habla de una *estructura de control iterativa*.

Estas tres estructuras de control – secuencia, condicional e iteración – son los tres pilares sobre los que se desarrolla la metodología denominada *Programación Estructurada*.

Entre los objetivos de esta asignatura, además de aprender a diseñar algoritmos correctos, se estableció la necesidad de hacerlos *legibles*. Para ello, se seguirán las normas establecidas por la programación estructurada. En esta introducción se considera conveniente justificar por qué es preciso seguir unas normas en el diseño del algoritmo, además de la necesidad (evidente) de que el algoritmo sea correcto. Es decir, por qué hay que aplicar una *metodología de programación* con el fin de facilitar tanto el diseño como el mantenimiento del *software*. Por eso, conviene realizar una pequeña reflexión sobre cómo ha evolucionado la informática y el trabajo de un programador en los últimos 60 años.

3.1.1. Evolución de los Computadores.

A pesar de que en siglos anteriores se pueden encontrar antecedentes de lo que hoy se conoce como computadores (la Máquina Aritmética de Pascal, el telar automático de Jacquard o el modelo teórico sobre una máquina pensante introducido por Charles Babbage y Lady Ada Lovelace), se suele situar la aparición de las primeras máquinas que se pueden definir como computadores electrónicos en la década de los años 40 del siglo XX (MARK I, ENIAC).

Sobre estas máquinas el programador debía utilizar el lenguaje de programación llamado "lenguaje máquina" o "lenguaje binario", un lenguaje cuyos únicos símbolos son el '1' y el '0', los cuales, combinados, forman secuencias que el computador es capaz de interpretar mediante un códi-

go[1]. Además de tener que utilizar un lenguaje muy incómodo, el programador disponía de máquinas realmente caras y lentas: eran realmente costosas en términos materiales (léase dinero) y en términos de prestaciones (léase velocidad de cálculo, capacidad de memoria, etc.). Como ejemplo, tómense las características del primer computador, el MARK I (1944): tenía 250.000 piezas, 800 Km. de cable, medía más de 15 metros de largo y cerca de 2,5 de alto y pesaba casi 5 toneladas. Necesitaba medio segundo para realizar la suma de dos números de 23 dígitos, 6 segundos para multiplicar dos números de 10 dígitos y 10 segundos para calcular su cociente. Su sucesor, el ENIAC (1946), mejoraba ya estas prestaciones (5000 sumas por segundo, por ejemplo).

Basta pensar en cualquiera de los ordenadores que hay hoy en día en el mercado (personales o portátiles, por ejemplo; ni que decir tiene que si la comparación se realiza con supercomputadores – para computación de altas prestaciones – aún resultaría más escandalosa), para comprender que a partir de ahí la tecnología evolucionó rápidamente. Además, también resulta fácil comprender que la tarea de un programador en esos días era muy difícil: un lenguaje de programación incómodo y máquinas con muy pocos recursos y muy caros. Por lo tanto, se aplaudía y se celebraba cualquier truco que le permitiera mejorar la velocidad de proceso, el tamaño del programa o, en general, cualquiera de las prestaciones físicas de la máquina.

Por fortuna, a medida que mejoraba la tecnología y se avanzaba en el desarrollo de programas, la tarea de los pioneros se simplificó. Desde el punto de vista del programador, el gran cambio surgió en la década de los 50-60, ya que aparecieron dos herramientas básicas en el desarrollo de programas: los *sistemas operativos* y los primeros *lenguajes de programación de alto nivel* (FORTRAN y COBOL). Los sistemas operativos son programas residentes en el computador que liberan al programador de tareas que nada tienen que ver con la resolución de un problema específico, sino con el propio funcionamiento de la máquina (la gestión de la comunicación del computador con el exterior, por ej.). Entre otras cosas, los sistemas operativos, facilitan la utilización de los lenguajes de programación de alto nivel, que no son otra cosa que lenguajes específicos para la comunicación con el computador, pero que no están basados en '1' y '0', sino en el lenguaje natural. Los lenguajes de programación permiten que la escritura de los programas sea más sencilla, pero es necesario que otro programa, el compilador o el intérprete lo traduzca a lenguaje máquina, el único que realmente entiende el computador.

En este nuevo escenario, por desgracia y aunque ya se disponía de herramientas de ayuda en el diseño de programas, los programadores siguieron trabajando sin ninguna metodología y recurriendo a los viejos trucos. Teniendo en cuenta que a medida que mejoran las prestaciones del computador (velocidad de proceso, capacidad de la memoria,...) los programas se van haciendo más grandes y complejos, el resultado fue que los programas se convirtieron en lo que se denomina "*bolas de spaghetti*": programas ilegibles para cualquier programador que no los haya escrito (y en ocasiones, también para su autor al cabo de un tiempo). Este hecho es tanto más lamentable si se tiene en cuenta que, tal y como se dijo en el tema 1, tras el *diseño* del programa y su puesta en marcha, viene una etapa igual de importante: la de *mantenimiento* de los programas. Es decir, resolver errores que aparecen a lo largo de su utilización, aumentar el número de operaciones que realiza, etc. Además, llega un momento en que mantener esta forma de trabajar se traduce en una pérdida de dinero. En la gráfica de la figura 3.1 se observa como ha evolucionado el coste de un soporte informático, desglosado en coste de hardware, diseño de software y mantenimiento de software: el mantenimiento se ha ido encareciendo hasta suponer más del 80 % del coste total.

Otro parámetro a tener en cuenta, es la necesidad de trabajar en equipo cuando se desarrollan programas realmente muy extensos (más de 100.000 líneas de instrucciones). Para coordinar estos equipos es necesario adquirir un cierto *estilo de programación*; ese estilo debe ser tal que prime la estructura del programa y su legibilidad por encima de todo. Sólo de esa forma puede realizarse un

[1]La justificación para utilizar este tipo de lenguaje es que el computador es una máquina electrónica; como tal lo más sencillo que puede entender es que le llegue corriente eléctrica, el '1', o que no le llegue, el '0'.

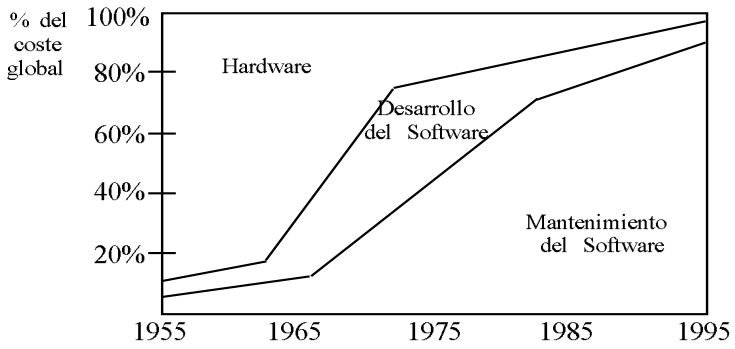


Figura 3.1: Evolución del coste de los sistema informáticos: las áreas representan el % de cada concepto.

buen mantenimiento del software y una cooperación eficiente entre equipos de trabajo. En 1968, nació el lenguaje de programación *Algol* (del cual se derivó el lenguaje *Pascal* en 1970), y un nuevo concepto, la *programación estructurada*. Fue el primer paso hacia las modernas metodologías de programación[2].

Por ello es importante que antes de desarrollar programas, se adquiera la idea de que es preciso adquirir ese estilo de programación. A continuación, se exponen sus principios básicos.

3.1.2. Programación Estructurada.

Definición 3.1 (Programación Estructurada) *Metodología o estilo de programación en la cual los programas se construyen mediante la combinación de un conjunto pequeño de* estructuras de control *bien utilizadas.*

El objetivo primordial al utilizar esta metodología es conseguir que el flujo de ejecución del código sea evidente a partir de la estructura sintáctica del texto del programa. Cuando se logra este objetivo se dice que se ha desarrollado un *programa estructurado*. Y la forma más sencilla de conseguirlo es respetar siempre la secuencia, leer una instrucción detrás de otra y en el mismo orden en que aparecen.

En otras palabras, la lectura del texto del programa debe permitir seguir con facilidad el desarrollo del mismo en los distintos casos que puedan plantearse durante su ejecución. A esto se le llama **legibilidad** del programa: el código no debe ser enrevesado y confundir al programador o al lector, debe entenderse qué hace y cómo lo hace. Este aspecto va dirigido no sólo al autor del código sino, fundamentalmente, al programador/lector que no lo ha diseñado y pretende interpretarlo.

Como se ha comentado anteriormente, el origen de la programación estructurada se encuentra en la denominada "crisis del software" ocurrida a finales de los años 60. Los lenguajes anteriores al lenguaje de programación Pascal, realizaban un uso intensivo de la instrucción *GoTo*. Esta instrucción provoca un salto incondicional a una determinada instrucción de un programa. Con ello se produce

[2]Por supuesto, ha habido una gran evolución en el diseño de software en estos años. Hay metodologías y herramientas mucho más sofisticadas que la Programación Estructurada (que supuso el punto de partida) para el desarrollo de grandes aplicaciones. Ello no quita que haya unos principios básicos que es conveniente aplicar (y unos hábitos de programación que es preciso adquirir) independientemente del tamaño de la aplicación a desarrollar.

una ruptura de la secuencia de instrucciones: la próxima instrucción a ejecutar no es la siguiente, sino que es una instrucción que puede estar 25 instrucciones más adelante (ó 100 instrucciones más atrás) de la última ejecutada. Su abuso fue en gran parte culpable de las llamadas "bolas de spaghetti", ya que un algoritmo o un programa con muchas instrucciones GoTo, rompe demasiado a menudo la secuencia de las instrucciones, dificultando la legilibilidad. Un punto clave de esta crisis, y que dio lugar a la programación estructurada, es el famoso artículo de Edsger Dijkstra titulado "*GoTo Statement Considered Harmful*" (La sentencia GoTo se considera perjudicial), publicado en Marzo de 1968 en Communications of the ACM [3].

En este artículo, Dijkstra expresaba su opinión de que la legibilidad de un código es inversamente proporcional al número de transferencias de control incondicionales (GoTos) que contiene. La idea es bastante evidente. Cuando, durante la lectura de un código se encuentra un GoTo, se debe interpretar del siguiente modo:

> "Por muy difícil que te haya resultado encontrarme en el flujo de control del programa (a través de qué instrucciones y circunstancias me has encontrado), detente ahora y encuentra la continuación del flujo lógico en un punto distinto (posiblemente lejano) del programa".

La programación estructurada dota al programador con estructuras de control de flujo en las que *sólo hay un posible punto de entrada y un posible punto de salida*. De esta forma se respeta la secuencia natural de las instrucciones que forman algoritmos y programas, lo que permite mejorar la legibilidad y, lo que es más importante, permitió también el desarrollo de herramientas para validar formalmente la corrección de los algoritmos.

3.1.3. El Concepto de Predicado. Cálculo de Predicados.

Tal y como se comentó al comienzo de esta introducción, la secuencia de instrucciones no permite, en general, describir una solución general de los problemas mediante algoritmos; para ello, es necesario tener la posibilidad de elegir, de entre varias secuencias de acciones, sólo una para su ejecución o de disponer de un mecanismo que permita repetir, un número finito de veces, la ejecución de una misma secuencia. En ambos casos, es necesario evaluar una condición o *predicado*.

Definición 3.2 (Predicado) *Sea $\mathcal{L}$ el conjunto* {`falso`, `cierto`}. *Dado un conjunto $\mathcal{E}$, una función proposicional o predicado es una aplicación de $\mathcal{E}$ en $\mathcal{L}$.*

Un predicado es, pues, un enunciado `cierto` para algunos elementos de $\mathcal{E}$ y `falso` para todos los demás. Por ejemplo, considérese el conjunto de los números naturales, $\mathcal{N}$, y el enunciado *"Dado x, tal que $x \in \mathcal{N}$, x es múltiplo de 3"*. Este predicado es una aplicación que toma valor cierto para algunos valores de x (3, 6, 9,...) y valor falso para otros (1, 2, 4, 5,...).

En un entorno algorítmico, los predicados tendrán como conjunto de partida al conjunto de valores tomados por uno o varios objetos del entorno. Por ejemplo, que el valor de la variable A sea 0, ó que el valor de la variable B sea positivo. Su forma coincide con la de las expresiones de tipo `BOOLE`, de ahí la importancia de este tipo y que se haya definido como uno de los tipos básicos, aún cuando no todos los lenguajes lo definan explícitamente.

[3] Encontraréis un enlace en la página web de la asignatura

Se distinguirán dos tipos de predicados:

Predicados elementales: son los formados por comparación de dos variables del mismo tipo mediante los operadores *relacionales* (==, $<$, $\leq$, $>$, $\geq$, $\neq$).

Las comparaciones entre objetos de tipo numérico (enteros y reales) se hace atendiendo al valor contenido en el objeto. Entre objetos de tipo carácter, se hace atendiendo al orden alfabético (se asumirá tal orden en todos los símbolos del objeto, tal y como se buscan palabras en un diccionario). En el tipo `BOOLE`, se suele asumir que `falso` $<$ `cierto`.

En aquellos lenguajes que lo soportan, se considera que los objetos de tipo `BOOLE` son predicados simples.

Predicados compuestos: son los formados por unión de predicados simples mediante *conectores* u operadores lógicos: `not` (ó $\neg$), `or` (ó $\vee$), `and` (ó $\wedge$)[4].

La evaluación se hace de acuerdo a la tablas de verdad del correspondiente operador.

Algunos ejemplos de predicados válidos en el cálculo de predicados son:

$$\begin{array}{cccc} A > B & C == (D + aux) & X == 1 \texttt{ or } (Z == 2 \texttt{ and } W > 3) & 8 == 10 \\ C + B \leq D & Valor \neq 0 & A > B \texttt{ and } B > C & \texttt{not } A \texttt{ or } B \end{array}$$

Existe un orden de evaluación, primero los predicados simples y, después, los compuestos. Para evaluar los compuestos se atiende a la prioridad asociada a los operadores lógicos. En cualquier caso, es recomendable utilizar paréntesis para indicar más claramente el orden de evaluación de un predicado.

Nótese, asimismo, que expresiones comunes en algunas ramas de las matemáticas como $A > B > C$, no son válidas dentro del cálculo de predicados. *Un predicado elemental sólo compara dos objetos*. La expresión anterior sería equivalente al predicado compuesto ($A > B$ `and` $B > C$). Esta regla es adoptada por casi todos los lenguajes de programación.

Como se ha dicho, en programación, hay muchas operaciones cuya realización depende de la evaluación de una condición, es decir, del valor de un predicado. Pero, además, los predicados constituyen la herramienta fundamental en el desarrollo de métodos formales, basados en la Lógica de Predicados de Primer Orden, LPPO, que permitan establecer si un algoritmo es o no válido. En este tema no se aborda la LPPO, pero se introducen algunos conceptos básicos sobre la lógica de las acciones en la que los predicados lógicos son especialmente importantes.

3.2. Esquemas Condicionales.

La secuencia es la estructura de control más simple y ya se ha manejado en los ejemplos del tema 2. En esta sección se inicia el estudio de las demás estructuras de control, presentando las estructuras de control condicional o esquemas condicionales.

[4]Existen más, pero en esta asignatura sólo se utilizarán éstos. En concreto, si se pretende desarrollar lógica de predicados de primer orden son imprescindibles tanto la implicación y la doble implicación como los cuantificadores existenciales y universales; pero, puesto que el tema de validez formal no se trata en la asignatura, se presentan sólo los conectores habituales en la escritura de algoritmos.

Definición 3.3 (Estructura de Control Condicional) *Una estructura de control condicional, o esquema condicional, permite expresar acciones que se ejecutarán (o no) dependiendo del resultado obtenido al evaluar un predicado.*

El primer esquema a considerar es el *esquema condicional simple* o *ejecución condicional*, que permite la opción de ejecutar o no una acción dependiendo del valor de un predicado. Responde a la siguiente formulación,

if (<predicado>) {
<instrucción>
}

La ejecución de este esquema es de la manera siguiente: se evalúa el predicado y si el resultado de su evaluación es `cierto`, se ejecuta la instrucción (o instrucciones) contenidas en el cuerpo del condicional (es decir, las indentadas y que se escriben entre llaves, { y }). Si no se cumple, se le ignora y se ejecuta la instrucción siguiente al condicional. Así, por ejemplo, dada la secuencia,

```
.....................
<instrucción1>;
if (<predicado>) {
     <instrucción2>;
     <instrucción3>;
}
<instrucción4>;
.....................
```

si se cumple el predicado (si el resultado de evaluarlo es `cierto`) la secuencia de instrucciones que ejecutaría el procesador es

```
.................
<instrucción1>;
<instrucción2>;
<instrucción3>;
<instrucción4>;
.................
```

Si no se cumple (el predicado es `falso`) la secuencia de instrucciones es,

```
.................
<instrucción1>;
<instrucción4>;
.................
```

Hay un segundo esquema condicional, el *esquema condicional doble* o *ejecución alternativa* que responde a la formulación

if (<**predicado**>**) {**
  <**instrucción1**>
}
else {
  <**instrucción2**>
}

En este caso, si se cumple el predicado se ejecuta la instrucción (o instrucciones) indentadas en el primer bloque de instrucciones (`<instrucción1>`). Si no se cumple, las instrucciones que ejecuta el procesador son las indentadas en el segundo bloque de instrucciones, el que se escribe después de la palabra reservada `else` (es decir, `<instrucción2>`). Por lo tanto, si se considera la secuencia,

```
.................
<instrucción1>;
if (<predicado>) {
    <instrucción2>;
}
else
{
    <instrucción3>;
}
<instrucción4>;
.................
```

caso de ser cierta la condición expresada en ⟨`predicado`⟩, la secuencia de instrucciones a ejecutar es la formada por la instrucciones

```
.............
<instrucción1>;
<instrucción2>;
<instrucción4>;
.............
```

Si no se cumple ⟨`predicado`⟩, la secuencia de instrucciones que ejecutará el procesador es

```
.............
<instrucción1>;
<instrucción3>;
<instrucción4>;
.............
```

3.2.1. Ejemplos del Uso de Esquemas Condicionales.

Ejemplo 1: Dados dos valores del mismo tipo, determinar cuál es el valor del máximo. Por ejemplo, sean `A` y `B` dos valores de tipo `ENTERO`.

Hay dos posibilidades para solucionar este problema. En la primera, la idea a desarrollar sería, simplemente, comparar ambos valores,

1. Si `A` es mayor que `B`, al resultado `maximo` se le debe asignar el valor `A`,
2. En caso contrario, al resultado `maximo` se le debe asignar el valor `B`.

El razonamiento conduce directamente al uso de un esquema condicional doble, y así se diseña la primera solución (nota: `ind.` servirá como abreviatura de `indeterminado`),

OBJETO	FUNCIÓN	V. I.	V. F.
`a`	Dato, de tipo `ENTERO`	`A`	`A`
`b`	Dato, de tipo `ENTERO`	`B`	`B`
`maximo`	Resultado, de tipo `ENTERO`	`ind.`	max(`A`, `B`)

```
algoritmo Maximo {
   /* Datos */
   int a, b;
   /* Resultado */
   int maximo;

   if (a > b) {
      maximo = a;
   }
   else {
      maximo = b;
   }
}
```

En esta solución, hay que hacer notar que la condición contraria a (`a` $>$ `b`) es (`a` $\leq$ `b`). ¿Qué ocurre si ambos valores son iguales?.

Una segunda posibilidad sería asignar inicialmente uno de los dos valores al resultado para comprobar después si esa asignación inicial era o no correcta,

1. Al resultado `maximo` se le asigna el valor `A`,
2. Se compara `maximo` con el valor `B`, y si el valor `B` es mayor se asigna a `maximo`.

En este caso basta con utilizar un condicional simple, puesto que no hay ninguna acción a realizar si la comparación (`b` $>$ `maximo`) resulta ser falsa. El entorno del algoritmo es el mismo que en el caso anterior.

```
algoritmo Maximo2 {
   /* Datos */
   int a, b;
   /* Resultado */
   int maximo;

   maximo = a;
   if (b > maximo) {
       maximo = b;
   }
}
```

Como en la primera solución, también en este algoritmo resulta de interés plantearse el caso de que ambos valores sean iguales.

Ejemplo 2: Un fabricante de papel vende las hojas de tamaño DIN A4 en paquetes de 500 hojas. El precio del paquete depende de la cantidad que se le compre: cuando es menor de 200 paquetes, cada uno se factura a 2 euros; si la cantidad está entre 201 y 500, cada paquete se factura a 1,75 euros y si son más de 500 los paquetes pedidos, el precio unitario es de 1,15 euros. Hay que calcular el importe a pagar correspondiente a un determinado pedido.

En este caso las ideas a desarrollar son las siguientes,

1. Si el pedido es de 200 o menos paquetes, el total resulta ser el número de paquetes por 2,
2. Si el pedido es de más de 200 y de menos de 501 paquetes, el total resulta ser el número de paquetes por 1,75,
3. Si el pedido es de más de 500 paquetes, el total resulta ser el número de paquetes por 1,15.

Estas ideas se pueden desarrollar de varias formas. La solución diseñada propone que la cantidad de paquetes pedidos se compare primero con el valor 200. Y, si es mayor, que se compare posteriormente con 500. De esta forma se determina el intervalo correcto de facturación.

OBJETO	FUNCIÓN	V. I.	V. F.
`cpp`	Dato, de tipo `ENTERO` que representa la cantidad de paquetes	cantidad pedida	cantidad pedida
`total`	Resultado, de tipo `REAL` que representa el importe total	ind.	total a pagar

```
algoritmo PrecioPedido {
   /* Dato */
   int cpp;
   /* Resultado */
   float total;

   if (cpp <= 200) {
      total = cpp * 2.0;
   }
   else {
      if (cpp <= 500) {
         total = cpp * 1.75;
      }
      else {
         total = cpp * 1.15;
      }
   }
}
```

En la solución se debe tener en cuenta que la condición contraria a (`cpp`$\leq$`200`) es (`cpp`>`200`). Por lo tanto, si hay que ejecutar la rama `else` del condicional seguro que el valor de `cpp` es mayor que 200. Por eso, basta con comparar sólo el valor de `cpp` con el valor `500` en el segundo esquema condicional: si el valor de `cpp` está comprendido entre `201` y `500` se ejecutará la primera rama y si es mayor que `500`, la segunda.

3.2.2. Esquemas Condicionales en C.

Lo que se ha expuesto hasta ahora es aplicable a todos los lenguajes de programación estructurados: todos poseen un esquema de *ejecución condicional*, `if`, y un esquema de *ejecución alternativa*, `if/else`.

Los esquemas condicionales se han presentado siguiendo una sintaxis que está basada en la del lenguaje `C`. En realidad, el `C` no obliga a indentar, pero la indentación es una de las primeras normas de estilo en programación para conseguir algoritmos legibles. Si sólo hay que ejecutar una instrucción en cada rama del condicional, `C` tampoco obliga al uso de llaves para delimitarla (aún cuando se hayan utilizado siempre en la exposición de los esquemas y en los ejemplos, como criterio propio de la asignatura). Es decir, cuando sólo hay una instrucción dentro del condicional, los dos esquemas se pueden escribir como

```
if (<predicado>)
  instrucción;
```

y

```
if (<predicado>)
  instrucción1;
else
  instrucción2;
```

En relación con la ejecución alternativa, el `C` dispone del *operador condicional*, "`? :` ". Con este operador se puede construir una *expresión condicional*, que tiene la forma general

(<predicado>) ? <expresión1> : <expresión2>

y que funciona de la siguiente manera: si el resultado de evaluar el predicado es `cierto`, se evalúa `<expresión1>` y el valor obtenido es el de la expresión condicional; y si `<predicado>` es `falso`, lo que se evalúa es `<expresión2>` y este es el valor de la expresión condicional.

Lo que se acaba de describir es cómo se evalúa una expresión condicional; en cuanto al uso, se usará como una expresión cualquiera, si bien suele ser especialmente útil cuando se pretende asignar un valor a una variable, optando entre dos mediante la evaluación de una condición. Por ejemplo, el cálculo del máximo de dos valores podría haberse escrito como:

```
maximo = (a > b) ? a : b;
```

y su ejecución sería equivalente a la del condicional doble utilizado anteriormente:

```
if (a > b) {
   maximo = a;
}
else {
   maximo = b;
}
```

Cuando las acciones a realizar son más complejas y dentro de la sentencia condicional se ha de ejecutar un bloque de instrucciones, entonces sí que hay que indicar donde empieza y donde acaba dicho bloque. Para ello se utilizan como delimitadores las llaves ({ y }):

```
if (<predicado>) {
   instrucción_1;
   instrucción_2;
   ...
   instrucción_N;
 }
```

o bien

```
if (<predicado>) {
   instrucción_11;
   instrucción_12;
   ...
   instrucción_1N;
 }
else {
   instrucción_21;
   instrucción_22;
   ...
   instrucción_2M;
 }
```

Se ha preferido seguir el criterio de utilizar siempre llaves para evitar confusiones, ya que en ocasiones es preciso utilizarlas para indicar el emparejamiento adecuado entre un determinado "`if`" y su correspondiente "`else`" (si es que hay varios). Por ejemplo: se quiere saber si un número es menor que `0` o si es igual a `1`; una posible secuencia de instrucciones en `C` podría ser la siguiente:

```
1 if (num > 0)
2   if (num == 1)
3     printf("num es igual a 1");
4 else
5   printf("num es menor que 0");
```

En `C` la indentación no es tenida en cuenta por el compilador. Este trozo de programa no hace lo que, aparentemente, se quiere. El "`else`" se asocia, si no se indica de alguna otra forma, al último "`if`" que "se ha abierto", el de la línea 2. El uso correcto de las llaves como delimitadores, permite asociar un "`else`" al "`if`" correspondiente y no al que por defecto le asigna el compilador. Así pues, para hacer lo que se pretendía en el anterior ejemplo, se han de utilizar las llaves:

```
if (num > 0) {
  if (num == 1)
    printf("num es igual a 1");
 }
else
  printf("num es menor que 0");
```

Existe otro esquema condicional que suele estar presente en los lenguajes de programación, si bien el funcionamiento y la estructura sintáctica no son tan uniformes como las de la ejecución condicional y la ejecución alternativa. Se trata de la *ejecución anidada*: como su nombre indica se trata de una sucesión de condicionales de dos ramas, anidados sucesivamente.

Para introducir este tipo de condicional, se propone una pequeña modificación del segundo de los ejemplos vistos anteriormente.

> *Sunpongamos que, por ser buenos clientes, conseguimos un nuevo descuento y, si el pedido es superior a los 800 paquetes, el precio del paquete es de 0.95 euros.*
>
> *Esto afectaría mínimamente a la solución diseñada, ya que basta con añadir el nuevo tramo:*

```
algoritmo PrecioPedido {
   /* Dato */
   int cpp;
   /* Resultado */
   float total;

   if (cpp <= 200) {
      total = cpp * 2.0;
   }
   else {
      if (cpp <= 500) {
         total = cpp * 1.75;
      }
      else {
         if (cpp <= 800) {
            total = cpp * 1.15;
         }
         else {
            total = cpp * 0.95;
         }
      }
   }
}
```

> *Ante la posible aparición de nuevos tramos, procederíamos de forma similar. Al analizar la estructura lógica de la solución puede verse que siempre se mantiene la misma pauta: sólo una de las condiciones será cierta y, en consecuencia, sólo una de las instrucciones se ejecutará.*

Cuando el flujo de ejecución de instrucciones obliga a realizar la ejecución alternativa y hay más de dos opciones, el uso de un condicional anidado simplifica la escritura del código, siempre que se esté completamente seguro de que los predicados que rigen cada rama son *excluyentes*, es decir, sólo es posible que uno sea cierto (y, entonces, seguro que todos los demás son falsos).

El lenguaje `C` no dispone de una estructura especial para este tipo de condicional, debiéndose utilizar secuencias anidadas de condicionales dobles, en los que es muy importante que se indique correctamente el emparejamiento entre un `if` y su correspondiente `else` (no hay que olvidar que el lenguaje tiene su propio criterio al respecto, si no se indica explícitamente).

Sólo para un caso particular de la ejecución anidada se dispone de una estructura especial, la estructura `switch`, cuya sintaxis es:

```
switch (<expresión>) {
  case <valor1> :
       <instrucción1>;
       break;
  case <valor2> :
       <instrucción2>;
       break;
  case <valor3> :
       <instrucción3>;
       break;
  ...
  case <valorN> :
       <instrucciónN>;
       break;
  default :
       <instrucción(N+1)>;
 }
```

El `switch` permite seleccionar un grupo de instrucciones de entre varios disponibles. La selección se basa en el valor resultante de evaluar `<expresión>`; dicho valor debe ser un valor entero (o bien de tipo `char`, ya que internamente `C` trata a los valores de este tipo como si fueran valores enteros).

Asociado a cada uno de estos valores hay asociado un bloque, de forma que si al evaluar `<expresión>` se obtiene el `<valorI>`, se ejecuta la instrucción (o secuencia de instrucciones) `<instrucciónI>`. Si al evaluar la `<expresión>` no se obtiene ninguno de los valores especificados, se ejecuta la instrucción, o instrucciones, asociadas a `default`.

Nótese, asimismo, que cada bloque de instrucciones, salvo el último, finaliza con `break`; esto es necesario para asegurar que sólo se ejecutará un único bloque, el seleccionado a partir del valor obtenido al evaluar la expresión que rige a esta estructura.

De esta forma, el funcionamiento descrito sería equivalente al siguiente fragmento de código,

```
if (<expresión>==<valor1>) {
    <instrucción1>;
}
else {
    if (<expresión>==<valor2>) {
```

```
        <instrucción2>;
    }
    else {
        if (<expresión>==<valor3>) {
            <instrucción3>;
        }
        else {
                        ...
                        if( <expresión>==<valorN> ) {
                            <instrucciónN>;
                        }
                       else {
                           <instrucción(N+1)>;
                       }
                      ...
        }
    }
}
```

Una aplicación típica de esta estructura es la construcción de programas en los que aparece un *menú de opciones*, es decir, aquellos en los que usuario debe optar por realizar una determinada operación entre varias posibles. Un ejemplo de este tipo de aplicaciones sería el menú de opciones de un cajero automático: el usuario puede optar entre varias operaciones (sacar dinero, ingresar, actualizar cartilla...) y una vez que ha elegido una, sólo una se realiza.

3.3. Esquemas Iterativos.

Definición 3.4 (Estructura de Control Iterativa) *Una estructura de control iterativa, más comúnmente denominada* iteración *o* bucle*, permite expresar la repetición de la ejecución de una instrucción, o una secuencia de instrucciones, dependiendo del resultado obtenido al evaluar un predicado.*

Existen dos posibilidades cuando hay que diseñar un bucle para resolver un problema: que sea necesario iterar la ejecución de determinada instrucción un *número conocido de veces* (por ejemplo, el número de sumas necesarias para calcular la media aritmética de 100 números), o que las acciones se repitan un número de veces variable y quizás *desconocido a priori*, y el bucle finalice su iteración tras evaluar una condición (por ejemplo, el número de divisiones necesarias para obtener la expresión en base 2 de un número finaliza[5] cuando se obtiene como cociente el valor 1).

Los lenguajes de programación estructurados permiten realizar ambos tipos de iteraciones mediante el bucle `while` o bucle *condicional*.

La forma general del bucle condicional es la siguiente,

while (<predicado>) {
  <instrucción>
}

[5]Se podría calcular a priori el número de veces que se repetirá la división, pero implica la necesidad de llevarse bien con los logaritmos y parece más sencillo utilizar una comparación (“cociente == 1?”) para gobernar el bucle.

La ejecución de un bucle `while` supone la evaluación del predicado de control que lo rige *ANTES* de ejecutar $\langle instrucción \rangle$. Por ello, es habitual que se le denomine *predicado de entrada*. Si el resultado de evaluar $\langle predicado \rangle$ es `falso`, finaliza la ejecución del cuerpo del bucle y se pasa a ejecutar la instrucción siguiente; si es cierto, se ejecuta la instrucción (o instrucciones) contenidas en el cuerpo del bucle, es decir, las indentadas y contenidas entre los delimitadores { y }. Al finalizar la ejecución completa de todas las instrucciones que forman el cuerpo del bucle, se vuelve a evaluar la condición (comienza de nuevo la ejecución).

Por lo tanto, el número de veces que se repite la ejecución de la secuencia de acciones es desconocido de antemano (depende de la evaluación del predicado de control) y puede variar desde 0 a cualquier número de veces.

3.3.1. Ejemplos del Uso de Esquemas Iterativos.

Ejemplo 1: Cálculo del cociente y resto de la división entera mediante restas sucesivas. Para ello se parte de la relación

```
dividendo = cociente * divisor + resto.
```

Para efectuar la división se restará el valor del divisor al dividendo tantas veces como sea posible (antes de que el dividendo se haga más pequeño que el divisor). El número de restas efectuadas será el cociente, mientras que el resultado de la última resta efectuada será el resto.

Por ejemplo, 34 = 5 * 6 + 4, o, lo que es lo mismo 34 / 6 = 5 y 34 % 6 = 4. Si se aplica el razonamiento anterior, se tiene:

34 - 6 = 28, 28 - 6 = 22, 22 - 6 = 16, 16 - 6 = 10, 10 - 6 = 4

En el momento en que se obtiene un resultado menor que 6, se deja de efectuar restas. En este caso, $4 < 6$; por lo tanto, el resto es 4 y, como se han realizado cinco restas mientras el resultado obtenido era mayor que el divisor, el cociente es 5.

Si se representan dividendo, dividor, cociente y resto mediante los objetos `dvdo`, `dvsor`, `coc` y `resto` respectivamente, las ideas a desarrollar son

1. Restar `dvsor` a `dvdo` mientras el resultado sea mayor o igual que `dvsor`,
2. El número de restas efectuadas será el cociente, `coc`,
3. El resultado de la última resta será el resto de la operación, `resto`.

OBJETO	FUNCIÓN	V. I.	V. F.
`dvdo`	Dato, de tipo `ENTERO` que representa el dividendo	`a` ≥ 0	`a` ≥ 0
`dvsor`	Dato, de tipo `ENTERO` que representa el divisor	`b` ≥ 0	`b` ≥ 0
`coc`	Resultado, de tipo `ENTERO` que representa el cociente	`ind.`	`a / b`
`resto`	Resultado, de tipo `ENTERO` que representa el resto	`ind.`	`a % b`

Con estas consideraciones se diseña el algoritmo `DivEntera`. Inicialmente se almacena el valor de `dvdo` en `resto`, y se va restando el valor de `dvsor` sucesivamente. El valor de `coc` se inicializa a 0 y se incrementa en una unidad por cada resta realizada, lo que permite utilizarlo como *contador* del número de restas efectuadas:

```
algoritmo DivEntera {
  /* Datos */
  int dvdo, dvor;

  /* Resultados */
  int coc, resto;

  resto = dvdo;
  coc = 0;
  while (resto >= dvsor) {
      resto = resto - dvsor;
      coc = coc + 1;
   }
}
```

Ejemplo 2: Cálculo de la suma de todos los números enteros comprendidos entre `1` y `n`. Para ello, se supone que se conoce el valor `n`, n$\geq$ 1, de un objeto entero, `num`. Puesto que hay que sumar todos los números entre 1 y `n`, se utilizará un objeto de tipo entero, `indice`, cuyo valor varíe entre el inicial, 1, y el final, `n`. Además, se necesitará otro objeto entero variable, `resultado`, que permita acumular la suma, al ir añadiendo a su valor, sucesivamente, cada uno de los valores que tome `indice`. Las ideas básicas serán:

1. Dado un valor de `indice` comprendido entre 1 y `n`, el siguiente valor se obtiene sumando 1 al valor ya almacenado,
2. Sea `resultado` con valor inicial 0; cada uno de los valores obtenidos en `indice` se acumula al valor ya almacenado en `resultado`,
3. El proceso acaba cuando `indice` toma el valor (`n`+1), que ya no se debe acumular.

OBJETO	FUNCIÓN	V. I.	V. F.
`num`	Dato, de tipo `ENTERO` que representa el máximo valor a sumar	`n` $\geq$ 1	`n` $\geq$ 1
`indice`	Objeto de tipo `ENTERO` que toma todos los valores entre `1` y `n`	`ind.`	`n`+1
`resultado`	Resultado, de tipo `ENTERO`	`ind.`	1+2+...+`n`

El resultado de las consideraciones efectuadas es el algoritmo `SumaEnteros`,

```
algoritmo SumaEnteros {
  /* Datos */
  int num;

  /* Resultados */
  int resultado;
```

```
  int indice;

  resultado = 0;
  indice = 1;
  while (indice <= num) {
      resultado = resultado + indice;
      indice = indice + 1;
   }
}
```

3.3.2. Potencial y Peligro de los Bucles Condicionales.

El bucle condicional es la herramienta más poderosa y versátil de la que dispone un programador.

Sin bucles no sería posible diseñar algoritmos completamente generales. Considérese de nuevo el problema del cálculo de la media aritmética, en la que varios números se suman y al final, se debe dividir el resultado acumulado entre la cantidad de números sumados. La primera cuestión que se hace evidente es que si no se dispone de un mecanismo que permita indicar que una operación básica, la suma, se repite varias veces, la escritura del algoritmo se haría tediosa: basta escribir el algoritmo que calcule la suma de 100 números, por ejemplo, sin utilizar bucles. Habría que escribir 99 veces la suma actuando sobre los distintos operandos.

Pero lo realmente interesante, no es resolver la media aritmética de 4, 100 ó 250 números. Si se pretende escribir algoritmos generales, hay que abordar el problema general, el cálculo de la media aritmética de *N números*. El método a seguir es siempre el mismo, sea cual sea la cantidad de números a sumar. Por lo tanto, el algoritmo correcto debe ser capaz de procesar cualquier cantidad de datos. Si no se dispusiera de la estructura iterativa, no se podría construir dicho algoritmo, ya que no habría forma de expresar que hay que realizar tantas sumas como sea necesario.

Este es el potencial del que dotan los bucles al programador. Por ello, es tan importante comprender bien cómo funcionan y cómo se diseñan. Se puede enunciar una propiedad de los bucles condicionales que puede ser útil a la hora de diseñar algoritmos,

Proposición 3.1 *Dado el esquema iterativo,*

$$\mathcal{W} \equiv \text{while } (\langle \textit{predicado} \rangle) \ \{ \ \langle \textit{instrucción} \rangle \ \}$$

cuando acaba la ejecución de $\mathcal{W}$ *se cumple* (`not` ⟨predicado⟩).

A veces es más sencillo pensar en la condición que debe cumplirse tras la ejecución del bucle. Al negarla, se obtiene el predicado de entrada.

Esta propiedad se puede demostrar sin más que parafrasear el esquema de funcionamiento de un bucle `while`: si mientras se cumple el predicado se está ejecutando la secuencia de instrucciones contenidas en el cuerpo del bucle, seguro que si ha finalizado la ejecución es porque ya no se cumple el predicado.

Pero este potencial, también supone un peligro. Puede que el predicado nunca se haga falso.

Al describir el funcionamiento de un bucle condicional, se puede acotar inferiormente el número de iteraciones que realizará: ninguna, si el predicado de entrada es falso la primera vez que se evalúa. Pero no es posible dar una cota superior; si el predicado es cierto y se comienza a iterar, no es posible, a priori, saber cuando dejará de cumplirse dicho predicado y, por lo tanto, cuando finalizará su ejecución. Ese es el riesgo que hay que asumir al disponer de un esquema que repite la ejecución de acciones dependiendo del valor de un predicado de control: los *bucles infinitos*.

Por lo tanto, al diseñar un bucle no sólo hay que comprobar que hace lo que se desea. Además, hay que comprobar que en algún momento su ejecución finalizará.

Un bucle infinito es un error lógico difícil de detectar, ya que el error no es sintáctico, es decir, una vez que se convierta el algoritmo en un programa, no sería detectado por el intérprete o por el compilador; el programa tampoco tendría errores de ejecución, del tipo que se suele producir cuando se intenta realizar una división entre cero, o una raíz cuadrada de un número negativo. Es más difícil de detectar en ejecución frente a otros errores de diseño que provoquen, por ejemplo, resultados erróneos, que pueden dar alguna pista sobre "qué funciona mal"... de hecho, es muy posible que no se llegue a ver ningún resultado, simplemente se produce una ejecución infinita (el programa "tarda" en mostrar los resultados, pero ¿es porque necesita un tiempo muy grande de ejecución o es porque se ha producido un bucle infinito?). El programa no acaba nunca y la única salida es la finalización por desconexión. Así, por ejemplo, la secuencia

```
i = 1;
while (i >= 1) {
  i = i + 1;
}
```

no contiene errores sintácticos, ni errores de ejecución; simplemente, su ejecución nunca finalizaría.

En general, la tarea de verificar que un bucle `while` no es infinito no es trivial, puesto que al depender el número de veces que se ejecuta del valor obtenido al evaluar una condición, sería preciso tener en cuenta todas las posibles combinaciones iniciales de las variables que intervienen en dicha condición antes de la ejecución del bucle. Existen mecanismos formales que permiten detectar y evitar un bucle infinito. Como primera medida se deben tener en cuenta las siguientes consideraciones en su diseño:

1. Se debe comprobar que funciona correctamente en una situación en la que la condición de entrada sea falsa.
2. Se debe comprobar que cada vez que el cuerpo del bucle se ejecuta, la nueva situación del entorno (después de la ejecución) es *similar* y *más simple* que la anterior situación (antes de la ejecución). Similar, en el sentido de que no haya cambiado drásticamente, y más simple, en el sentido de que quede menos trabajo por realizar que en la iteración anterior (quedan menos iteraciones que ejecutar para que finalice la ejecución del bucle, porque se consigue que el estado del entorno se vaya transformando para conseguir que el predicado deje de cumplirse).
3. Teniendo en cuenta 1) y 2), se puede concluir que el bucle se ejecutará de forma correcta sin importar el número de veces que se tenga que ejecutar; de ambas condiciones se desprende que las instrucciones a ejecutar dentro del cuerpo del bucle conducen a que la condición se haga falsa y que alguna vez acabará su ejecución.

3.3.3. Bucles con Contador.

Todos los lenguajes de programación disponen de una estructura de control especial para aquellos casos en que se conoce el número de veces que se debe ejecutar una instrucción a priori. Se le suele conocer como *bucle con contador*.

Un bucle con contador no tiene el mismo poder computacional que un bucle condicional; de hecho, el bucle con contador es un caso particular de este. Es decir, todo lo que se puede hacer con un bucle con contador se puede hacer con un bucle condicional. Pero el recíproco no es cierto.

Tiene una sintaxis más sencilla y puede ser más cómodo de usar. Además, bien usado, ofrece otra ventaja: no hay posibilidad de caer en un bucle infinito.

En el siguiente fragmento de algoritmo se ofrece un ejemplo de su uso. Es una secuencia de instrucciones alternativa para calcular la suma de los `n` primeros números enteros:

```
resultado=0;
for (indice=1; indice<=num; indice=indice+1) {
   resultado=resultado+indice;
}
```

En este problema es posible usar un bucle con contador, `for`, porque se sabe a priori cuántas veces hay que repetir el proceso, basta con dar un valor a `num`. Si se compara este algoritmo con el diseñado anteriormente, se observa que en este ya no hay que actualizar el valor de `indice`: un bucle `for` gestiona el valor del contador, en este caso `indice`, de forma que desde el valor inicial, `1`, alcance el valor final, `num`, incrementando automáticamente su valor en cada iteración, en una unidad.

El contador es una variable de tipo entero. Algunos lenguajes de programación permiten su definición mediante otros tipos, si bien siempre son tipos relacionados con el tipo `ENTERO`.

El esquema general del bucle `for` es el siguiente:

for (v=vi; v<=vf; v=v+vp) {
<instrucción>
}

El funcionamiento general del bucle consiste en la repetición de la instrucción (o instrucciones) contenidas en el cuerpo del bucle tantas veces como se pueda añadir al valor inicial de `v` el valor del *paso* o incremento, `vp`, partiendo del valor inicial, `vi`, sin que se sobrepase el valor final, `vf`. Es decir, la primera iteración se realiza con `v = vi`; la segunda, con `v = vi+vp` y así, sucesivamente, hasta alcanzar el valor final. En general, el valor de `v` variará según la fórmula

$$\texttt{v} = \texttt{v} + \texttt{vp}, \quad \texttt{vi} \leq \texttt{v} \leq \texttt{vf}.$$

El valor del paso, `vp`, permite definir cómo se desea que varíe el contador desde el valor inicial hasta el final. Si lo que se desea es que en cada iteración la variable contador incremente su valor de 2 en 2, por ejemplo, entonces se le daría a `vp` el valor 2; o, si lo que se desea es contar hacia atrás – que el valor inicial de `v` sea `10` y el final `1`, por ejemplo,– se escribiría

```
for (v=10; v>=1; v=v-1) {
   <instrucción>
}
```

Un bucle `for` siempre se puede escribir utilizando un bucle `while`. Para ello hay que distinguir dos casos, según el valor del paso, `vp`,

1. Si el valor del paso es positivo:

   a) El esquema general del bucle `for` puede sustituirse por un bucle `while` de la siguiente forma:

   ```
   v=vi;
   while (v<=vf) {
     <instrucción>
     v=v+vp;
   }
   ```

   Por lo tanto, si el valor inicial de la variable de control es superior al valor final, no se entra en el bucle.

   b) El número de ejecuciones de $\langle instrucción \rangle$ es

   `(vf-vi+1)/vp`.

2. Si el valor del paso es negativo:

   a) El esquema general del bucle `for` puede sustituirse por un bucle `while` de la siguiente forma:

   ```
   v=vi; /* vi debe ser mayor que vf */
   while (v>=vf) {
     <instrucción>
     v=v+vp; /* vp es negativo */
   }
   ```

   Por lo tanto, si el valor inicial de la variable de control es inferior al valor final, no se entra en el bucle.

   b) El número de ejecuciones de $\langle instrucción \rangle$ es

   `(vi-vf+1)/(-vp)`.

Con esto se pone de manifiesto que no es posible que un bucle `for` pueda tener una ejecución infinita, puesto que el fin de su ejecución está asegurada.

3.3.4. Esquemas Iterativos en C.

La notación utilizada en las subsecciones anteriores es la propia de `C`. Sólo queda recalcar, pues, algunas peculiaridades del lenguaje.

En `C` existen los bucles `while` y `for`, tal y como se han visto en la exposición anterior. Con respecto a la sintaxis hay que comentar que, al igual que se hizo al presentar los esquemas condicionales, se ha optado por delimitar siempre el cuerpo de los bucles utilizando llaves. El lenguaje `C` no obliga a utilizarlas cuando sólo se quiere ejecutar una instrucción. Otra sintaxis de uso habitual en la escritura de los bucles `for`, consiste en el uso del *operador de incremento* (`++`), para incrementar el contador en una unidad; es decir, se suele utilizar la expresión `i++` en lugar de `i=i+1`. Ambas son equivalentes a todos los efectos.

Pero la diferencia más significativa se refiere a que, en `C`, todos los bucles son condicionales. Por supuesto, lo es el bucle `while`, el condicional por excelencia. Pero también lo puede ser el

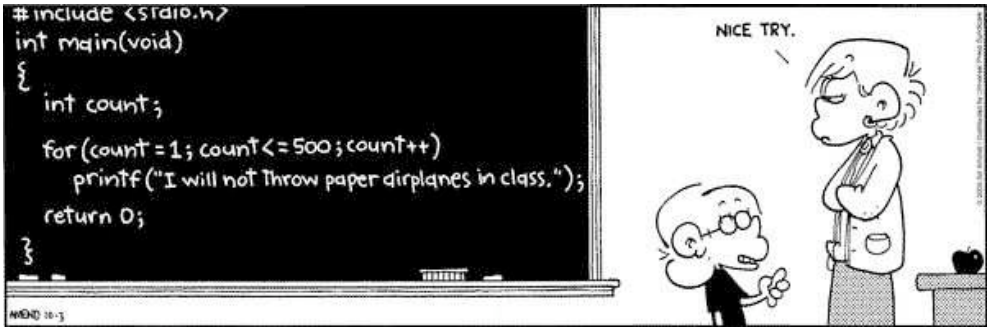
bucle `for`. Esto ocurre porque, en este lenguaje, el bucle `for` es una estructura iterativa en la que se pueden definir tres partes[6]:

- `inicio`, una instrucción que se ejecuta **una sola vez, antes de empezar** la ejecución del cuerpo del bucle.
- `condición`, que se evalúa antes de ejecutar las instrucciones del cuerpo del bucle en cada iteración: si es cierta se ejecutan las instrucciones de la estructura y, si no lo es, se pasa a ejecutar la instrucción siguiente.
- `incremento` (o `decremento`), que se ejecuta al acabar las instrucciones del cuerpo del bucle y antes de evaluar la condición del mismo.

La sintaxis completa en `C` es la que se muestra a continuación:

```
for (inicio;condición;incremento) {
  instrucción_1;
  instrucción_2;
  ...
  instrucción_N;
}
```

La estructura presentada en la sección anterior como *bucle con contador* es una simplificación de la que `C` permite, y es la que se utilizará en la asignatura.



Un error bastante común al trabajar en `C`, consiste en poner un "`;`" al final de la definición de una estructura repetitiva (ya sea un `while` o un `for`). El compilador entiende que se quiere repetir la "instrucción vacía", no las instrucciones asociadas al bucle. En el siguiente ejemplo se repite 10 veces "nada" (se ejecuta 10 veces la línea 1) y luego se visualiza el valor final de `i` (se ejecuta una vez la línea 2, cuyo efecto es escribir 11), en lugar de visualizar los 10 primeros números, que era lo que se pretendía (es decir, ejecutar 10 veces la línea 2).

```
1 for (i=1;i<=10;i=i+1);
2   printf("%d", i);
```

El error puede ser aún más grave si ocurre con un bucle `while`, puesto que se habría introducido un bucle infinito (ya que el contador `i` del ejemplo no se incrementa: el bucle empieza y acaba en la línea 2, y el incremento de `i` está fuera, en la línea 5), en el que no se hace nada:

[6]Cada una de las partes de la estructura `for` puede estar formada por más de una instrucción, separadas por comas, e incluso puede estar vacía. A efectos de esta asignatura se utilizará siempre una sola instrucción en cada una de las partes.

```
1 i=1;
2 while (i<=10);
3 {
4   printf("%d", i);
5   i=i+1;
6 }
```

El lenguaje C dispone de un esquema iterativo más, el `do while`. Su funcionamiento es similar al del `while`, con una salvedad: en el `while` la evaluación del predicado de control se realiza antes de la ejecución del cuerpo del bucle (por ello, se tenía que el número de ejecuciones del bucle podía ser 0). Pero, a veces, es deseable que el predicado se evalúe DESPUÉS de la ejecución del cuerpo del bucle.

> *Se desea leer un carácter de teclado, de forma que el usuario opte entre tres valores posibles; además, no se permite que el usuario introduzca un valor no válido, por lo que se repetirá el proceso de pedir un valor hasta que sea uno que corresponda a una opción correcta. Este proceso se puede realizar mediante la secuencia,*

```
printf("Introduzca opción elegida (A, B o C):");
scanf("%c", &opcion);
while (opcion != 'A' && opcion != 'B' && opcion != 'C') {
  printf("Introduzca opción elegida (A, B o C):");
  scanf("%c", &opcion);
}
```

En el ejemplo anterior, claramente, hasta que no se lee el valor de `opcion` no es posible evaluar el predicado; por eso hay que efectuar una lectura antes de poder ejecutar el bucle `while`. El `do while` permite que la lectura de datos no tenga que realizarse fuera del bucle,

```
do {
  printf("Introduzca opción elegida (A, B o C):");
  scanf("%c", &opcion);
} while (opcion != 'A' && opcion != 'B' && opcion != 'C');
```

ya que primero se ejecutan las instrucciones que forman el cuerpo del bucle y después se evalúa el predicado: si es `cierto`, se vuelve a ejecutar el cuerpo del bucle y si es `falso` finaliza la ejecución del bucle.

En general, el bucle `do while` tiene la siguiente sintaxis:

do {
  <**instrucción**>
} while (<**predicado**>**);**

Del comportamiento descrito se sigue, además, que `<instrucción>` se ejecuta al menos una vez. Por lo tanto, si bien el funcionamiento es similar, no se puede considerar equivalente a un bucle `while`.

3.4. Lógica de las Estructuras de Control.

Se definió en el tema 1 una acción como *un suceso capaz de modificar el entorno en el que se aplica*. Por lo tanto, esta modificación se puede caracterizar describiendo el entorno antes y después de la ejecución de la acción considerada.

Para ello, se pueden formular dos predicados, uno que describa un estado posible del entorno antes de la ejecución de la acción, y otro que describa el nuevo estado del entorno después de la ejecución de la acción.

Pueden existir varios estados posibles del entorno antes de la ejecución de una acción, ya que un estado viene dado por la ejecución de acciones anteriores; pero *una vez fijado el antecedente y la acción, el consecuente debe ser único*.

Por ejemplo,

$$\underset{antes}{/ * x == 4 * /} \quad \underset{acción}{\texttt{x} = \texttt{x} + 2} \quad \underset{después}{/ * x == 6 * /}.$$

El uso de estos predicados será más significativo cuánto más precisa sea la descripción que realizan del estado del entorno.

Por ejemplo, una descripción del entorno como la anterior,

$$/ * x == 4 * / \quad \texttt{x} = \texttt{x} + 2 \quad / * x == 6 * /$$

es mucho más precisa que la que se expone a continuación

$$/ * x \leq 6 * / \quad \texttt{x} = \texttt{x} - 4 \quad / * x \leq 2 * /,$$

y bastante más descriptiva que la que se muestra en el siguiente caso,

$$/ * x \neq 0 * / \quad \texttt{x} = \texttt{x} * 4 \quad / * x \neq 0 * /,$$

Cuanto más fuerte sea el predicado utilizado en la descripción del entorno, tanto más útil será. En este contexto, un predicado es más fuerte que otro si hay menos valores que lo hagan cierto.

Con esto se obtiene una mera descripción, con lo que el mecanismo no se diferencia mucho de una traza. Una posibilidad más interesante resulta al considerar la idea de *especificar* la acción, y no solamente describirla.

Si sólo se describe el estado del entorno, es posible tener la siguiente situación,

$$/ * x == \mathcal{X} \; and \; y == \mathcal{Y} * /$$
$$\texttt{x} = \texttt{x}/\texttt{y}$$
$$/ * x == \mathcal{X}/\mathcal{Y} \; and \; y == \mathcal{Y} * /$$

Sin embargo, no sería del todo cierta, ya que no se está describiendo completamente la situación tras la ejecución de la acción: hay otra posibilidad, la de un error de ejecución. ¿Por qué no introducir en la descripción inicial de la situación del entorno una condición que garantice que no se provocará ese error?. Es decir, ¿por qué no especificar en que condiciones esa acción puede ejecutarse sin error?

$$/ * x == \mathcal{X} \; and \; y == \mathcal{Y} \; and \; \mathcal{Y}! = 0 * /$$
$$\texttt{x} = \texttt{x/y}$$
$$/ * x == \mathcal{X}/\mathcal{Y} \; and \; y == \mathcal{Y} * /$$

Definición 3.5 (Precondición y Postcondición) *Una precondición es un predicado que debe satisfacerse antes de la ejecución de una acción. Una postcondición es un predicado que debe satisfacerse después de la ejecución de la acción.*

La precondición y la postcondición son dos tipos de predicados conocidos como **asertos** y se utilizan para expresar predicados **que deben satisfacerse** exactamente en el instante en el que la ejecución del algoritmo los alcance. Se formularán como un predicado lógico delimitado por los separadores de comentario, /* */, y se utilizará, en general, la notación

$$/ * \textbf{precondición} * / \quad \textbf{acción} \quad / * \textbf{postcondición} * /.$$

No son recursos de los lenguajes de programación, sino que son herramientas lógicas que pueden utilizarse tanto para demostrar la corrección formal de un algoritmo (una vez que éste ha sido ya diseñado), como para ayudar en el diseño: si se sabe qué debe cumplirse antes (precondición) y qué debe cumplirse después (postcondición), se tienen indicios sobre cuál es la acción a desarrollar entre ambos asertos.

En la notación anterior con el genérico `acción` se especifica cualquier proceso. Por simplicidad, todos los ejemplos que se han utilizado hasta ahora eran asignaciones, pero por `acción` se debe entender tanto la acción básica (asignación), como cada una de las tres estructuras de control (secuencia, condicional y bucle). A continuación, se discutirá cada uno de estos casos. Y, en el próximo tema, se ampliará el mecanismo a la especificación de algoritmos completos.

En concreto, cuando la acción a especificar es una *asignación*, existe una relación básica entre la precondición y la postcondición que se debe satisfacer,

$$/ * \langle \texttt{expresión} \rangle \; satisface \; precondición * /$$
$$\langle \texttt{var} \rangle = \langle \texttt{expresión} \rangle;$$
$$/ * \langle \texttt{var} \rangle \; satisface \; postcondición * /$$

Lógica de una Secuencia.

Si lo que se desea es especificar una *secuencia*, la especificación es correcta si se encuentra una especificación encadenada de cada una de las acciones que la forman.

Por ejemplo, considérese la secuencia `A` formada por tres acciones `a`, `b` y `c`; si la precondición de cada acción, excepto la primera, coincide con la postcondición de la que le precede, se tiene entonces una regla de deducción que permite escribir:

$$\textbf{si se cumple} \begin{array}{lll} /*P*/ & \texttt{a} & /*Q_1*/ \\ /*Q_1*/ & \texttt{b} & /*Q_2*/ \\ /*Q_2*/ & \texttt{c} & /*Q*/ \end{array}, \textbf{entonces se cumple } /*P*/\ \texttt{A}\ /*Q*/$$

Esta relación se expresa como $P \stackrel{A}{\Rightarrow} Q$: del estado *P* se deriva el estado *Q* al realizar la acción *A*. La especificación de *A*,

$$/*P*/ \quad A \quad /*Q*/$$

es correcta si se consigue encontrar dicha secuencia encadenada de predicados.

De la relación anterior, se desprende que *dos secuencias de instrucciones tienen ejecución equivalente si admiten la misma especificación*.

Ejemplo: El problema de intercambiar el valor de dos variables queda especificado mediante la precondición

$$/*x == \mathcal{X}\ and\ y == \mathcal{Y}*/$$

y la postcondición

$$/*x == \mathcal{Y}\ and\ y == \mathcal{X}*/.$$

La secuencia que se propone es la siguiente,

```
aux = x;
x = y;
y = aux;
```

Falta verificar si cumple la especificación, es decir, si a partir de la precondición se alcanza la postcondición:

$/*x == \mathcal{X}\ and\ y == \mathcal{Y}*/$
`aux = x;`
$/*x == \mathcal{X}\ and\ y == \mathcal{Y}\ and\ aux == \mathcal{X}*/$
`x = y;`
$/*x == \mathcal{Y}\ and\ y == \mathcal{Y}\ and\ aux == \mathcal{X}*/$
`y = aux;`
$/*x == \mathcal{Y}\ and\ y == \mathcal{X}\ and\ aux == \mathcal{X}*/$

Efectivamente, se llega a una situación que satisface la postcondición. De hecho, incluye una condición más, $aux == \mathcal{X}$; `aux` *es una variable necesaria para desarrollar la solución, si bien no interviene en la especificación del proceso. Se volverá a insistir sobre esta idea en el próximo tema, al hablar de la especificación de algoritmos.*

3.4.1. Lógica del Esquema Condicional.

En la especificación de un esquema condicional hay que tener en cuenta que la ejecución es alternativa y que, por lo tanto, dependiendo de que el predicado que lo rige sea o no sea cierto se ejecutará una de las dos ramas y sólo una; evidentemente, el estado final del entorno tras la ejecución de cada una de las dos ramas puede ser distinto. Así, expresar la lógica del esquema condicional

$$\mathcal{C} \equiv \texttt{ if (c) \{a\} else \{s\}}$$

es afirmar que el predicado `c` es `cierto` antes de ejecutar la instrucción {`a`} y `falso` antes de ejecutar la instrucción {`s`}.

Entonces, si el condicional tiene el predicado /*P*/ como precondición y además se cumple

$$\begin{array}{rcl} /*\ P\ and\ \texttt{c}\ */ & \texttt{a} & /*\ Q_1\ */ \\ /*\ P\ and\ not(\texttt{c})\ */ & \texttt{s} & /*\ Q_2\ */, \end{array}$$

el condicional se puede especificar como

$$\begin{array}{rcl} /*\ P\ and\ \texttt{c}\ */ & \mathcal{C} & /*\ Q_1\ */ \\ /*\ P\ and\ not(\texttt{c})\ */ & \mathcal{C} & /*\ Q_2\ */. \end{array}$$

A partir de esta relación, se pueden estudiar tres propiedades básicas del esquema condicional: la ejecución equivalente, la simplificación por abajo y la simplificación por arriba.

Proposición 3.2 (Ejecución Equivalente) *Dados los esquemas condicionales*

$$\mathcal{C}1 \equiv \texttt{if}\ (\texttt{c})\ \{\texttt{a}\}\ \texttt{else}\ \{\texttt{s}\}$$
$$\mathcal{C}2 \equiv \texttt{if}\ (\texttt{not}(\texttt{c}))\ \{\texttt{s}\}\ \texttt{else}\ \{\texttt{a}\}$$

su ejecución es equivalente.

Demostración:

En efecto, si se supone que se cumple

$$\begin{array}{rcl} /*\ P\ and\ \texttt{c}\ */ & \texttt{a} & /*\ Q_1\ */ \\ /*\ P\ and\ not(\texttt{c})\ */ & \texttt{s} & /*\ Q_2\ */, \end{array}$$

de acuerdo a la relación anterior, se puede afirmar que la especificación del esquema $\mathcal{C}1$, es

$$\begin{array}{rcl} /*\ P\ and\ \texttt{c}\ */ & \mathcal{C}1 & /*\ Q_1\ */ \\ /*\ P\ and\ not(\texttt{c})\ */ & \mathcal{C}1 & /*\ Q_2\ */. \end{array}$$

Pero, además, por lo que respecta a $\mathcal{C}2$, se tendrá

$$\begin{array}{rcl} /*\ P\ and\ not(\texttt{c})\ */ & \mathcal{C}2 & /*\ Q_2\ */ \\ /*\ P\ and\ not(not(\texttt{c}))\ */ & \mathcal{C}2 & /*\ Q_1\ */ \end{array} \Rightarrow /*\ P\ and\ \texttt{c}\ */\ \mathcal{C}2\ /*\ Q_1\ */.$$

Puesto que `not(not(c))` da como resultado el predicado `c`, la conclusión es que ambos esquemas son equivalentes, ya que tienen la misma especificación.

c.q.d.

Según esto los condicionales,

`if (x%3==0) {`
`  x=x/3;`
`}`
`else {`
`  x=2*x+1;`
`}`
	`if (x%3!=0) {`
	`  x=2*x+1;`
	`}`
	`else {`
	`  x=x/3;`
	`}`

son de ejecución equivalente.

Proposición 3.3 (Simplificación por Abajo) *Dado el esquema condicional*

$$\mathcal{C} \equiv \texttt{if (c) \{a1; a3\} else \{a2; a3\}}$$

y la secuencia $\mathcal{A}$ formada por la instrucciones

$$\mathcal{A} \equiv \begin{cases} \mathcal{C}1 \equiv & \texttt{if (c) \{a1\} else \{a2\}} \\ & \texttt{a3;} \end{cases}$$

el efecto de ejecutar cualquiera de las dos es equivalente.

Demostración

Si se supone que se cumple

$$\begin{array}{rcccc} /*\, P\ and\ \texttt{c}\, */ & \texttt{a1} & /*\, Q_{11}\, */ & \texttt{a3} & /*\, Q_1\, */ \\ /*\, P\ and\ not(\texttt{c})\, */ & \texttt{a2} & /*\, Q_{21}\, */ & \texttt{a3} & /*\, Q_2\, */, \end{array}$$

se puede afirmar lo siguiente sobre el esquema condicional $\mathcal{C}$,

$$\begin{array}{rccc} /*\, P\ and\ \texttt{c}\, */ & \mathcal{C} & /*\, Q_1\, */ \\ /*\, P\ and\ not(\texttt{c})\, */ & \mathcal{C} & /*\, Q_2\, */. \end{array}$$

Al estudiar la lógica de la secuencia $\mathcal{A}$ se obtiene,

$$\begin{array}{rcccc} /*\, P\ and\ \texttt{c}\, */ & \mathcal{C}1 & /*\, Q_{11}\, */ & \texttt{a3} & /*\, Q_1\, */ \\ /*\, P\ and\ not(\texttt{c})\, */ & \mathcal{C}1 & /*\, Q_{21}\, */ & \texttt{a3} & /*\, Q_2\, */. \end{array}$$

o, lo que es lo mismo,

$$\begin{array}{rccc} /*\, P\ and\ \texttt{c}\, */ & \mathcal{A} & /*\, Q_1\, */ \\ /*\, P\ and\ not(\texttt{c})\, */ & \mathcal{A} & /*\, Q_2\, */. \end{array}$$

por lo tanto, ambos esquemas son equivalentes.

c.q.d.

Es decir, las dos secuencias siguientes tiene ejecución equivalente:

`if (x%3==0) {`
`  x=x/3;`
`  i=i+1;`
`}`
`else {`
`  x=2*x+1;`
`  i=i+1;`
`}`
	`if (x%3==0) {`
	`  x=x/3;`
	`}`
	`else {`
	`  x=2*x+1;`
	`}`
	`i=i+1;`

Proposición 3.4 (Simplificación por Arriba) *Dado el esquema condicional*

$$\mathcal{C} \equiv \texttt{if (c) \{a1; a2\} else \{a1; a3\}}$$

y la secuencia A *formada por la instrucciones*

$$\mathcal{A} \equiv \begin{cases} & \texttt{a1;} \\ \mathcal{C}1 \equiv & \texttt{if (c) \{a2\} else \{a3\}} \end{cases}$$

el efecto de ejecutar cualquiera de las dos NO siempre será EQUIVALENTE.

Demostración

Si se supone que se cumple

$$\begin{array}{rcccc} /*\, P\ and\ \texttt{c}\, */ & \texttt{a1} & /*\, Q_{11}\, */ & \texttt{a2} & /*\, Q_1\, */ \\ /*\, P\ and\ not(\texttt{c})\, */ & \texttt{a1} & /*\, Q_{21}\, */ & \texttt{a3} & /*\, Q_2\, */, \end{array}$$

se puede afirmar lo siguiente sobre el esquema condicional $\mathcal{C}$,

$$\begin{array}{rccc} /*\, P\ and\ \texttt{c}\, */ & \mathcal{C} & /*\, Q_1\, */ \\ /*\, P\ and\ not(\texttt{c})\, */ & \mathcal{C} & /*\, Q_2\, */. \end{array}$$

Al estudiar la lógica de la secuencia $\mathcal{A}$ se obtiene,

$$\begin{array}{rcccc} /*\, P\ and\ \texttt{c}\, */ & \texttt{a1} & /*\, Q_{11}\, */ & \mathcal{C}1 & /*???\, */ \\ /*\, P\ and\ not(\texttt{c})\, */ & \texttt{a1} & /*\, Q_{21}\, */ & \mathcal{C}1 & /*???\, */. \end{array}$$

No es posible afirmar nada sobre el comportamiento del condicional $\mathcal{C}$ cuando su antecedente es $/*\, Q_{11}\, */$ o $/*\, Q_{21}\, */$, a no ser que la ejecución de la acción `a1` verifique que el predicado `c` sigue cumpliéndose. Como esta situación no está garantizada en general, no siempre se podrá realizar una simplificación por arriba en un esquema condicional.

c.q.d.

Las dos secuencias siguientes tiene ejecución equivalente:

```
if (x%3==0) {
   i=i+1;
   x=x/3;
}
else {
   i=i+1;
   x=2*x+1;
}
```

```
i=i+1;
if (x%3==0) {
   x=x/3;
}
else {
   x=2*x+1;
}
```

ya que la ejecución de la instrucción que se ha sacado fuera del condicional no afecta a la evaluación del predicado `(x%3==0)`*. Sin embargo, en el ejemplo siguiente,*

```
if (x>=a) {
   x=2*x;
   a=x+c;
}
else {
   x=2*x;
   a=3*x+d;
}
```

```
x=2*x;
if (x>=a) {
   a=x+c;
}
else {
   a=3*x+d;
}
```

la ejecución no es equivalente, ya que la asignación que se ha eliminado del condicional afecta a la evaluación del predicado. Considérese, por ejemplo, que los valores iniciales de `a` *y* `x`*, fueran 5 y 3, respectivamente.*

3.4.2. Lógica del Esquema Iterativo.

El último caso a considerar es la especificación de los bucles. En principio, para determinar la postcondición se puede utilizar la propiedad ya conocida,

> "Dado el esquema iterativo,
>
> $$\mathcal{W} \equiv \textit{while}\ (\langle\text{predicado}\rangle)\ \{\ \langle\text{instrucción}\rangle\ \}$$
>
> cuando acaba la ejecución de $\mathcal{W}$ se cumple (`not`($\langle$predicado$\rangle$))."

Es decir, seguro que (`not`($\langle$predicado$\rangle$)) forma parte de la postcondición, cuya especificación dependerá, además, de las acciones a desarrollar en el cuerpo del bucle.

Para comprobar la validez del diseño hay que asegurar que, además de hacer lo que debe, el bucle finaliza. Ya se ha introducido el problema de los bucles infinitos y se han dado una serie de consideraciones a tener en cuenta en el diseño para intentar evitar este riesgo.

Existen herramientas que permiten demostrar formalmente la correción de un bucle, es decir, que hace lo que debe y que su ejecución no es infinita. No sólo hay que estudiar su antecedente y su consecuente,

```
/*P*/
while (<predicado>) {
   <instrucción>
}
/*Q*/
```

sino lo que ocurre *dentro* del bucle; la herramienta formal que se utiliza en este caso es un aserto que se denomina *invariante*.

Concepto de Invariante.

Definición 3.6 (Invariante de un Bucle) *Predicado que se debe cumplir antes y después de la ejecución completa del bucle y al principio y al final de cada una de las iteraciones del bucle.*

El uso de invariantes no sólo permite asegurar la validez de un bucle (acaba y hace lo que debe); al igual que precondiciones y postcondiciones, también facilita el diseño del bucle. Como ejemplo, considérese el *"Problema de la Lata de Judías"*,

El problema de la Lata de Judías:

Una lata de judías contiene judías blancas y judías pintas (y, además, se dispone de un almacén aparte de judías pintas en cantidad ilimitada); se debe repetir tantas veces como se pueda el siguiente proceso:

- *sin mirar, tomar dos judías de la lata.*
- *si son del mismo color, tirarlas y meter en la lata una judía pinta (para ello disponemos del almacén de judías pintas).*
- *si son de distinto color, se vuelve a colocar la blanca dentro de la lata y se tira la judía pinta.*

La ejecución de este proceso reduce en uno el número de judías contenidas en la lata. La repetición debe terminar con exactamente una judía en la lata, ya que entonces no se podrá coger un par de judías. La cuestión es la siguiente:

¿Se puede saber el color de esa última judía, basándose sólo en el número inicial de judías de cada color?

Si se aborda la solución del problema por "prueba y error", la tarea será muy pesada, ya que hay que considerar todos los casos posibles (1 blanca, 1 pinta; 2 blancas, 1 pinta; 1 blanca, 2 pintas; 2 blancas, 2 pintas, etc.).

Tal vez exista una propiedad simple, relacionada con el número de judías, que permanezca inalterable a medida que se realiza el proceso, y que, junto con el hecho de que sólo permanece una judía al final de éste, nos ayude a encontrar la respuesta; es decir, una propiedad invariante.

Supongamos que al finalizar el proceso la judía es pinta. ¿Qué propiedad es cierta tras la finalización y se puede generalizar para que sea un invariante?. Uno es un número impar; quizás la imparidad del número de judías pintas permanezca siempre. Pero no, ya que de hecho el número de judías pintas cambia de par a impar en cada movimiento. Otra propiedad que se cumple al finalizar el proceso es que el número de judías blancas es par (0); y de hecho, cada movimiento o bien elimina dos judías blancas o bien no elimina ninguna. Por lo tanto, la paridad del número de judías blancas es una propiedad invariante. Si, por contra, suponemos que la última judía es blanca, aplicando un razonamiento similar, se llega a que el número de judías blancas siempre es impar.

Esto permite dar la siguiente solución al problema: si el número de judías blancas es par, la última judía es pinta; si el número de judías blancas es impar, la última judía es blanca.

Un invariante es otro tipo de aserto, como las precondiciones y postcondiciones. Por lo tanto, también el enunciado del invariante toma la forma de un predicado lógico.

Sea, por ejemplo, el problema de calcular la suma de todos los números enteros entre ***1*** *y* ***n****, que se resolvería con la secuencia*

```
.............
x = 1 ;
i = 1 ;
while (i < n) {
    i = i + 1;
    x = x + i ;
}
.........
```

se puede enunciar el siguiente invariante,

$$/* x = \sum_{j=1}^{i} j */,$$

es decir, al principio y al final de cada iteración, antes de realizar la iteración y después de haberla ejecutado se cumple que en `x` *se almacena el valor de la suma de los números enteros entre* `1` *e* `i`.

Un bucle puede tener más de un invariante. De todos los invariantes de un bucle interesan los que permitan demostrar su corrección al asegurar que se satisfacen las dos condiciones siguientes:

1. Al finalizar la ejecución del bucle se alcanza el consecuente Q (y, por lo tanto, es correcto al cumplir la especificación).
2. El bucle ejecuta un número finito de iteraciones.

Para comprobar que se cumple la primera condición, se utiliza la siguiente propiedad:

Teorema 3.1 *Dado un bucle $\mathcal{W}$ y un invariante $\mathcal{I}$ para $\mathcal{W}$, si la ejecución de $\mathcal{W}$ acaba, lo hace en un estado en el que se cumple ($\mathcal{I}$* `and (not` ⟨`predicado`⟩`)`*), siendo* ⟨`predicado`⟩ *la condición de entrada del bucle $\mathcal{W}$.*

La demostración se basa en un razonamiento similar al siguiente: el invariante se cumple al final de cada iteración; por lo tanto, también al final de la última. En la última, además, se debe cumplir `not <predicado>`.

En el ejemplo anterior, cuando `i=n`*, en* `x` *se habrá acumulado la suma de todos los enteros de* `1` *a* `n`*.*

Además, se cumple `not <predicado>`*, ya que* `n<n` *es falso.*

Por lo tanto, al salir del bucle se cumple

$$(x = \sum_{j=1}^{n} j) \texttt{ and } (\texttt{not}(\texttt{n} < \texttt{n})).$$

Esto se puede expresar como

$$(\mathcal{I} \texttt{ and } (\texttt{not}\langle\texttt{predicado}\rangle)) \overset{\mathcal{W}}{\Rightarrow} \text{Q},$$

relación que puede ser útil en la verificación (dado el invariante comprobar qué consecuente se alcanza) o en el diseño (fijada la postcondición, encontrar el invariante).

En lo que respecta a la segunda condición, para demostrar que un bucle tiene fin se debe calcular una función limitadora.

Definición 3.7 *Una función limitadora es una función entera, t, tal que*

1. *si se cumple ($\mathcal{I}$* `and` $\langle$`predicado`$\rangle$*), entonces $t > 0$ (está acotada inferiormente por el cero) y*
2. *su valor decrece en cada iteración del bucle.*

Si se encuentra, se podrá asegurar que el bucle termina.

Siguiendo con el ejemplo anterior, se considera la función $t = n - i$. Si se cumple la condición

$$(x = \sum_{j=1}^{i} j) \texttt{ and } (\texttt{i} < \texttt{n}).$$

se ve que, efectivamente, $t > 0$. Además, en cada iteración el valor de `i` *se incrementa,*

$$i + 1 > i\,,$$

por lo tanto, el valor de t se decrementa

$$n - (i + 1) < n - i\,.$$

Se puede concluir que el bucle no será infinito.

En resumen, se puede dar el siguiente esquema para verificar un bucle:

```
/*P*/
<iniciación del bucle>;
/*I*/
while (<predicado>) {
   /*I and <predicado>*/
   <instrucción>;
   /*I*/
}
/*I and (not <predicado>)*/
/*Q*/
```

lo que conduce a las siguientes etapas de verificación (el bucle termina y es correcto) :

1. El invariante es cierto inmediatamente antes de entrar en el bucle.
2. El invariante se mantiene en cada iteración, es decir, es cierto inmediatamente antes de comenzar la ejecución del cuerpo del bucle y es cierto inmediatamente después de finalizarla.
3. La finalización del bucle lleva al consecuente.
4. Hay una función limitadora que permite concluir que el bucle no es infinito.

3.5. Resumen y Consideraciones de Estilo.

Con los contenidos del tema 2 y del tema 3 se dispone ya de las herramientas básicas para construir algoritmos de cierta entidad. De hecho, en el resto de la asignatura no se presentará ninguna nueva estructura de control; simplemente, además de aprender a organizar este código básico mediante la definición de funciones y procedimientos, se procederá a estudiar cómo organizar los datos, de forma que su proceso sea lo más efectivo posible.

En relación a los objetivos marcados en el tema 1, en especial en lo referido a las etapas a seguir en el diseño de algoritmos, los contenidos del tema 3 están relacionados con,

1. *Entender el problema*: Identificar para cada problema cuáles son los datos y cuáles los resultados es básico y el primer paso para entender qué se ha de hacer. Esta pauta se ha mantenido en todos los ejemplos de los temas 2 y 3. Además, en el tema 3 se ha formalizado este mecanismo mediante la introducción del concepto de *precondición* y *postcondición*, que en el tema 4 se generalizará para funciones y procedimientos.
2. *Plantear y planificar la solución*: Todavía no se han introducido esquemas generales y aún no se ha hecho mucho hincapié desde el punto de vista metodológico. Sin embargo, la especificación formal introducida al final del tema 3 debería ayudar a reflexionar sobre el código. Es interesante que esos conceptos, utilizados de una manera algo más informal de como se han introducido en teoría, ayuden a formular correctamente comentarios que, a la vez que documenten el código, especifiquen partes especialmente significativas en el desarrollo de la solución.
3. *Formular la solución*: Ya se conoce la sintaxis y el significado de cada uno de los elementos básicos en la construcción de algoritmos: los tipos básicos, la asignación, la construcción y uso de expresiones y, además, las tres estructuras básicas de la programación estructurada: la secuencia, el condicional y la iteración.

4. *Evaluar la corrección*: Los asertos, en forma de comentarios, deberían ayudar a reflexionar sobre el código. No interesará tanto formularlos formalmente; para alcanzar los objetivos de la asignatura bastará con una formulación simple pero que obligue a razonar sobre los algoritmos.

5. *Implementar la solución*: A lo largo de los temas 2 y 3 se han introducido también los conceptos básicos del lenguaje `C`. Junto con el trabajo que se desarrollará en las prácticas 1, 2 y 3, se dispone ya de un buen número de herramientas prácticas para desarrollar los programas.

Con respecto a la última etapa y siguiendo la misma línea que en el tema 2, se introducen las siguientes consideraciones sobre la adquisición de un buen estilo de programación:

- De nuevo, insistir en la necesidad de documentar los programas mediante el uso de comentarios. Es conveniente relacionar esta facilidad con la especificación de secuencias, condicionales o bucles que sean especialmente significativos en el desarrollo del código, o especialmente complicados.
- La indentación consiste en introducir espacios a la izquierda de las órdenes, de forma que las instrucciones que forman una estructura (una secuencia, un condicional, un bucle) queden alineadas y visualmente "incluidas" dentro de un mismo bloque. Es una buena costumbre utilizarla y permite mejorar la claridad de los programas. El compilador no tiene en cuenta la indentación (los bloques de instrucciones se delimitan en `C` por las llaves y no por la indentación), y sólo se utiliza a efectos de mejorar el mantenimiento. Pero conviene no olvidar que el propio autor del código también puede mantenerlo y sufrir los efectos "perniciosos para la salud" de un código mal documentado y mal indentado.

3.6. Glosario.

cuerpo de (condicional/bucle): bloque de instrucciones en cada una de las ramas de un condicional o en el interior de un bucle. Cada ejecución supone la ejecución completa de todas y cada una de las instrucciones del cuerpo. Normalmente se escribirán indentadas y entre delimitadores.

delimitador: Cada uno de los dos símbolos especiales que, en pareja, indican el principio y el final de una porción especial de código. Por ejemplo, `{` y `}` para indicar principio y fin de bloque de instrucciones y `/*` y `*/` para indicar principio y fin de comentario.

especificación: En programación, se entenderá como la reformulación del enunciado de un problema, de forma que se indica de forma precisa cuál es el estado inicial del entorno y cuál el estado final que se debe alcanzar.

indentar: Estilo de escritura en la que un párrafo tiene sus márgenes incluidos dentro de los de otro párrafo principal. En nuestro caso, sirve para delimitar visualmente un bloque de instrucciones con respecto a la estructura que las gobierna.

traza: Descripción exhaustiva del estado del entorno (esto es, del valor de todas las variables) que se suele realizar después de la ejecución de todas y cada una de las instrucciones del algoritmo.

variable contador: Variable cuyo valor cambia de forma regular recorriendo un rango de valores entre un determinado valor inicial y un determinado valor final.

variable acumulador: Variable sobre la que se realizan sucesivas operaciones de la misma clase. Suele ser típico acumular sumas: en ese caso, se inicializa al valor 0, y su valor se incrementa sucesivamente añadiendo a su valor el resultado de sumar una determinada cantidad.

3.7. 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. "Curso de Programación", Jorge Castro et al., McGraw-Hill. 1993.
4. "El Lenguaje de Programación C. Diseño e Implementación de Programas". Félix García, Jesús Carretero, Javier Fernández & Alejandro Calderón. Pearson Education, Madrid 2002.
5. "The C Programming Language". Brian W. Kernigham & Dennis M. Ritchie. Prentice-Hall Inc. 1988.

3.8. Actividades y Problemas Propuestos.

1. Sean a y b dos variables enteras con valores 3 y 5, respectivamente. Indicar si las siguientes condiciones son ciertas o falsas:

```
a + b < 10
!((a + b) <= 10)
a >= 2 && b >= 5
a < 10 || (a > 0  &&  b < 0)
a > 0 && (b < 2 || !(b > 5))
a < 5 && b < 8
!(a >= 5 || b >= 8)
```

2. Sean A, B, C y D variables enteras. Se considera la secuencia

```
if (((A > 0) || (B > C)) && ((D > A) || (D < 5))) {
   A = 0 ;
   D = B + C ;
}
else {
   C = A - B ;
   if (C < 0){
      D = -D ;
   }
   B = 0 ;
}
```

   Determinar los valores finales para las variables, sabiendo las siguientes posibilidades de valores iniciales:

   a) A = 5, B = 3, C = 4, D = 6

   b) A = -1, B = 3, C = 4, D = 3

   c) A = -1, B = -2, C = 4, D = 3

3. Sean X, Y y S variables enteras. Dada la secuencia

```
if (X < 0)
  S = 1 ;
else
    if (Y > 0)
        S = 1 ;
    else
       S = 0 ;
```

escribir una secuencia equivalente, sin anidamiento de condicionales y sin repetir la instrucción S = 1.

4. Dada la secuencia

```
if (p && q)
     a ;
else
     s ;
```

donde `p` y `q` son predicados elementales y `a` y `s` son acciones, escribir una secuencia equivalente, en la que sólo se evalúe cada vez un predicado elemental.

5. ¿Son equivalentes las tres secuencias siguientes?

```
s=0;
if (p) {
   s=3;
}
else {
   if (!q) {
      s=3;
   }
}
```

```
s=0;
if (!(!p&&q)) {
     s=3;
}
```

```
s=0;
if (p || !(q)) {
     s=3;
}
```

6. A, B y C son tres variables enteras. Escribir una secuencia de instrucciones que determine cuál es la variable de mayor valor y devuelva el resultado en una variable MAX.

7. Dados tres valores enteros A, B y C, escribir una secuencia de instrucciones que los ordene en orden creciente.

8. Las siguientes secuencias hacen lo mismo, pero ¿cuál es mejor y por qué? (nota: los valores de a, b y c son distintos dos a dos.)

```
if ((a>b) && (a>c)) {
  max = a;
}
if ((b>a) && (b>c)) {
  max = b;
}
if ((c>a) && (c>b)) {
  max = c;
}
```

```
if ((a>b) && (a>c)) {
  max = a;
}
else {
  if ((b>a) && (b>c)) {
    max = b;
  }
  else {
    if ((c>a) && (c>b)) {
      max = c;
    }
  }
}
```

9. Dado el siguiente fragmento de código, reescribirlo utilizando un único condicional, que además esté simplificado al máximo:

```
if (b < 10) {
    if (b >= 5)
         a = a*2;
    if (b < 5) {
       if  (b >= 0)
           a = a*2;
    }
}
```

10. Escribir una secuencia de instrucciones que determine a cuánto asciende la factura de la luz de un abonado. Para ello, se conoce AI, antiguo índice y NI, nuevo índice, que representan los valores leídos en el contador de la luz. El resultado se quiere sobre la variable IMPORTE, que se calcula sabiendo que un abonado

    a) Paga 5 euros por gastos fijos de contrato,

    b) El consumo se determina por tramos: los primeros 100 Kws, a 5 céntimos el Kw; los 150 Kws siguientes, a 3 céntimos el Kw; si el consumo excede de 250 Kws, esa fracción se cobra a 2 céntimos el Kw.

11. Reescribir la siguiente secuencia en forma de un único condicional regido por un predicado compuesto, tan simple como sea posible. Justificar todos los cambios realizados.

```
if (a > b)
    if (b-a > 0)
         a = a + 1;
    else
        if (x > a)
              if (x > b)
                   if ((2*x) > (a+b))
                        x = 0;
                   else
                      if (x < 100)
                            x = 1;
```

12. Simplificar al máximo las siguientes secuencias, justificando las acciones realizadas, sabiendo que las variables X, Y, I y K, son enteras:

`if (X>0) {`
`  I=I+2;`
`  K=0;`
`  Y=2*X;`
`}`
`else {`
`  I=I+2;`
`  K=0;`
`  Y=0;`
`}`
	`if (X==3) {`
	`  X=Y;`
	`  Y=0;`
	`  I=X+Y;`
	`}`
	`else {`
	`  X=Y;`
	`  Y=K*2;`
	`  I=X+Y;`
	`}`

13. Las secuencias siguientes ¿hacen lo mismo?

`s=0;`
`x=a;`
`s=b;`
`if (x>=c) {`
`  x=x%c;`
`}`
`else {`
`  x=x+1;`
`}`
	`s=b;`
	`if (x>=c) {`
	`  x=a;`
	`  x=x%c;`
	`}`
	`else {`
	`  x=a;`
	`  x=x+1;`
	`}`

14. ¿Verdadero o Falso (justificar)?: La ejecución de cada una de las dos secuencias siguientes es **exactamente** igual.

`if (v<V1) {`
`  <inst_1>;`
`}`
`else {`
`  if (v>V2) {`
`  <inst_2>;`
`  }`
`}`
	`if (v<V1) {`
	`  <inst_1>;`
	`}`
	`if (v>V2) {`
	`  <inst_2>;`
	`}`

15. Antes de ejecutarse el siguiente fragmento de código se cumple el predicado

/* j < N and acum == 0 */.

Analizar el fragmento e indicar que ocurriría al ejecutarlo:

```
i=j;
while (i<=N) {
   if (j<i) {
      i=i-1;
   }
   else {
      i=i+1;
   }
   acum=acum+i*i;
}
```

16. Simplificar al máximo la secuencia siguiente, justificando adecuadamente todos los cambios realizados:

```
/* x e y son variables reales */
/* n, entero, n=N>1 */
y = x;
for (i=1; i<=n; i=i+1){
   if (i<n) {
      y = i*x;
   }
   else {
      y = 2*x;
   }
}
```

17. Dado el siguiente esquema condicional:

```
if (<cond1>){
  if (<cond2>){
    <inst1>;
  }
  else {
    if (<cond3>){
      <inst2>;
      <inst3>;
    }
```

```
  }
}
else {
  if (<cond3>){
    <inst2>;
    <inst3>;
  }
}
```

¿Es posible substituirlo por un único condicional? ¿Por qué? Dar una versión equivalente con el menor número posible de condicionales anidados.

18. Simplificar al máximo el siguiente condicional, sabiendo que `a`, `b`, `c` y `d` son enteros:

```
a = 3;
if (a < 3) {
   b = 2*a;
}
else {
   if (b > c) {
      a = a + b;
      b = b + c;
      c = c + d;
   }
   else {
      a = a + b;
      b = b + c;
      c = 2*c;
      c = c + d;
   }
}
```

19. ¿Verdadero o Falso (justificar)?: "Los dos condicionales siguientes siempre producirán los mismos resultados",

```
if (p) {
   <inst_1>;
}
else {
   <inst_2>;
}
```

```
if (p) {
   <inst_1>;
}
if (!p) {
   <inst_2>;
}
```

20. ¿Verdadero o Falso (justificar)?: "Los dos condicionales siguientes son equivalentes",

```
if (p) {
   for(j=k;j<=q;j++) {
      b[i]=a[j];
      i++;
   }
}
else {
   for(k=j;k<=m;k++) {
      b[i]=a[k];
      i++;
   }
}
```

```
if (p) {
   for(j=k;j<=q;j++) {
      b[i]=a[j];
   }
}
else {
   for(k=j;k<=m;k++) {
      b[i]=a[k];
   }
}
i++;
```

21. Simplificar al máximo el siguiente condicional:

```
if (a>=b) {
   a=a/2;
   if (a>=c) {
      a=a/2;
   }
   else {
      a=a/2;
   }
}
else {
   a=a/2;
}
```

22. Dado el siguiente bucle, en el que `i` y `t` son enteros,

```
/* n es un entero, n>1 */
t=1;
i=0;
while (t<n) {
  t=t*2;
  i=i+1;
}
```

indicar para cada uno de los predicados siguientes si puede ser o no un invariante y por qué:

a) { 2^i<= n && n>0 }

b) { n > 0 && i<=log_2n }

23. Simplificar al máximo la siguiente secuencia de código:

```
/* Pre: j==V1, V1 > 0 */
   int aux, i;
   aux=0;
   i=0;
   while (i<=j) {
      if (i>=j) {
         aux=aux+(i*j);
         i=i+1;
      }
      else {
         aux=i*j;
         i=j;
      }
   }
   printf(" %d\n", aux);
}
```

24. Dada la variable entera NUM, que contiene un valor n $\geq$ 1, elaborar una secuencia de instrucciones que determine la suma de los `n` primeros números naturales.

25. Dado un entero i, i $>$ 0, escribir una secuencia de instrucciones que nos devuelva el menor entero n, tal que $2^n > i$.

26. Escribir una secuencia de instrucciones que determine el cociente y el resto de la división entera de dos números enteros A y B. (Nota: mediante restas y sumas).

27. Escribir una secuencia de instrucciones para calcular x^n, siendo n un número entero.

28. Dado un valor $n \geq 1$, elaborar una secuencia de instrucciones que calcule $n!$.

29. Escribir un programa que haga lo siguiente: Se irán leyendo valores reales por teclado y el objetivo es detectar secuencias.

    Una secuencia es una serie de números consecutivos iguales. Por ejemplo, en la siguiente serie hay 6 secuencias:

    ```
    0.25, 0.5, 0.5, 1.75, 0.1, 0.1, 0.1, 0.1, 0.15, 0.8, 0.8, 0.0
    ```

    Cuando se lea el valor `0.0` finaliza la introducción de números y se deberá escribir cuántas secuencias se han detectado.

30. La sucesión de Fibonacci se obtiene de acuerdo a

$$fibonacci(n) = \begin{cases} 1 & si\ n = 1, \\ 1 & si\ n = 2, \\ fibonacci(n-1) + fibonacci(n-2) & si\ n > 2 \end{cases} .$$

    Escribir una secuencia de instrucciones que calcule el número de Fibonacci asociado a un entero n.

31. Escribir una secuencia de instrucciones que permita calcular el término n de la sucesión

$$\begin{aligned} &s_0 = 1 \\ &s_1 = 1 \\ &s_2 = 1 \\ &s_n = s_{n-2} + s_{n-3},\ n \geq 3. \end{aligned}$$

    Es decir, los tres primeros términos son 1, 1, 1; el siguiente se calcula sumando los términos penúltimo y antepenúltimo. Si, por ejemplo, n=12, los términos a calcular serían: 1, 1, 1, 2, 2, 3, 4, 5, 7, 9, 12, 16, 21,... y habría que devolver el valor 21.

32. Escribir una secuencia de instrucciones que calcule el término k-ésimo de la sucesión

$$\begin{aligned} &T(0) = 1 \\ &T(1) = 1 \\ &T(n) = T(n-1) + (n-1) * T(n-2),\ n \geq 2. \end{aligned}$$

33. Escribir una secuencia de instrucciones que convierta una variable entera en otra, también entera, en la que el entero original aparezca del revés. Por ejemplo, el entero 357 se debe convertir en el 753.

34. Escribir una secuencia de instrucciones que dado un número entero lo reduzca a la suma de sus dígitos, de forma que el resultado sea un número de un sólo dígito. Por ejemplo, dado el número 13674891,

$$13674891 \rightarrow 1+3+6+7+4+8+9+1 = 39 \rightarrow 3+9 = 12 \rightarrow 1+2 = 3$$

    el resultado pedido es 3.

35. Escribir una secuencia de instrucciones que calcule la exponencial de un número real `a`, de acuerdo a la serie,

$$e^a = \sum_{n=0}^{\infty} \frac{a^n}{n!} = 1 + a + \frac{a^2}{2} + \frac{a^3}{3!} + \frac{a^4}{4!} + \ldots + \frac{a^n}{n!} + \ldots$$

    Aproximar el resultado hasta que para algún `k` se cumpla que $a^k/k! \leq 10^{-5}$.

36. Escribir una secuencia de instrucciones que calcule la exponencial de `a` según la serie del problema 35, pero aproximando hasta que k=20.

37. La función seno se puede calcular de acuerdo a la serie

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

mientras que para calcular la función coseno se utiliza la serie,

$$\cos x = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$$

Dado un real, `x`, escribir una secuencia de instrucciones que calcule el seno y el coseno de `x` simultáneamente utilizando estas series y de la forma más eficiente posible. Se considerará una aproximación válida que $|\sin^2 x + \cos^2 x - 1|$ sea menor que 10^{-6}.

38. Escribir una secuencia de instrucciones que permita determinar si un número es perfecto. Un número es perfecto si es resultado de la suma de sus divisores propios; por ejemplo, el 28 tiene como divisores propios al 1, al 2, al 4, al 7 y al 14, y se cumple que

$$28 = 1 + 2 + 4 + 7 + 14.$$

39. Escribir una secuencia de instrucciones que indique si dos números son amigos. Dos números son amigos si la suma de los divisores del primer número (excluido él) es igual al segundo número y viceversa, es decir, la suma de los divisores del segundo número (excluido él) es igual al primer número.

    Por ejemplo, 220 y 284 son amigos. Los divisores de 220 son 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 y 110, que suman 284 y los divisores de 284 son 1, 2, 4, 71 y 142, que suman 220.

40. Un número odioso es un número que, cuando se pasa a base 2, tiene un número impar de `1`'s. Por ejemplo, el 1, el 2 (`10`), el 4 (`100`), el 7 (`111`), el 8 (`1000`), el 11 (`1011`), ...

    Hay que escribir una secuencia de instrucciones que, dado un valor entero K, devuelva cuántos números odiosos hay entre 1 y K.

41. Se dice que un número es apocalíptico si contiene entre sus dígitos la secuencia 666. Por ejemplo, el 16667234, el 6660, el 34666, el 66666666, el 10266663... El 1346786956, por ejemplo, NO lo es: los '6' no forman secuencia.

    Escribir una secuencia de instrucciones que, dado un número entero, indique si dicho número es o no es apocalíptico.

42. Escribir una secuencia de instrucciones que permita calcular el máximo común divisor de dos números, basándose en el método de Euclides:

$$MCD(u,v) = \begin{cases} u & si\ u = v, \\ MCD(u, v-u) & si\ u < v, \\ MCD(u-v, v & si\ u > v \end{cases} .$$

43. El método de multiplicación de enteros a la rusa se puede enunciar de la siguiente forma: el multiplicando se divide (división entera) entre 2 hasta que el resultado sea 1; y tantas veces como se divide el multiplicando se multiplica por 2 el multiplicador. El resultado se obtiene

al sumar todos los números que aparecen en la columna del multiplicador que correspondan a un número impar en la columna del multiplicando. Por ejemplo,

$$45 * 19 = 855$$

45	19
22	38
11	76
5	152
2	304
1	608

$$19 + 76 + 152 + 608 = 855$$

Escribir una secuencia de instrucciones que permita multiplicar dos números mediante este método.

44. Dados dos objetos variables de tipo entero, `x` e `y`, tales que sus valores son positivos y que el valor de `x` es mayor o igual que el valor de `y`, y dado el bucle,

```
/* x>=y && y>0 */
v=0;
while (y>0) {
   x=x-1;
   y=y-1;
   v=v+1;
}
```

¿cuál de las siguientes expresiones elegirías como candidato a invariante? ¿Por qué?

a) { v == x - y }

b) { x >= y && y >= 0 }

45. Dado el bucle

```
/* P>0 */
k=0;
m=P;
while (m>1) {
   m=m/2; /* División Entera */;
   k=k+1;
}
```

¿cuál de los siguientes predicados será un invariante del bucle? Justificar.

a) $\{2^k <= P \;\&\&\; P > 0\}$

b) $\{k = log_2 P \;\&\&\; P > 0\}$

46. Dado el siguiente bucle, en el que `i`, `n` y `N` son enteros:

```
i=0;
n=N;
while (n>0) {
  i=i+1;
  n=n/10;
}
```

de entre los dos predicados siguientes ¿cuál es un invariante? ¿Por qué?

a) { n == N/(10^i) }

b) { n >= 0 && i==log_{10}N }

47. Dado el siguiente bucle, en el que `resta`, `x` e `y` son enteros:

```
resta=0;
while (x > y) {
  x = x-1;
  resta = resta+1;
};
```

de entre los dos predicados siguientes ¿cuál es un invariante? ¿Por qué?

a) { resta <= | x - y |}

b) { x >= 0 && y >= 0 && resta == x-y }

48. Si podéis, echadle un vistazo a la introducción del libro "The Science of Programming" de David Gries, *Why Use Logic?*.

Son tres páginas muy sencillas de leer (aunque esté escrito en inglés) y con un ejemplo muy simple, describe como dos programadores van descubriendo la utilidad de incorporar asertos en su código.