Capítulo 4

Introducción al Diseño Descendente

Índice General

"Hey, girl, I got something
I think you ought to know."

Led Zeppelin. "Communication Breakdown".

4.1. Acciones No Primitivas. Concepto de Parámetro.

Se propone la siguiente secuencia de instrucciones para resolver el problema de ordenar tres valores enteros:

```
max = A;
med = B;
min = C;
if (med > max) {
  Intercambiar med y max;
}
if (min > max) {
  Intercambiar min y max;
}
if (min > med) {
  Intercambiar min y med;
}
```

En esta secuencia se ha utilizado una expresión que no pertenece a un lenguaje de programación: ningún lenguaje contempla la acción *Intercambiar (...) y (...)*. Sin embargo, su uso en la secuencia no convierte la secuencia en ininteligible. Más bien al contrario, aunque nadie haya definido formalmente qué significa esa acción, se puede tener la idea intuitiva de que consistirá en que dos objetos intercambien sus valores. Si, además, se realiza la siguiente especificación:

/* x == $\mathcal{X}$ && y == $\mathcal{Y}$ */
`Intercambiar x e y`
/* x == $\mathcal{Y}$ && y == $\mathcal{X}$ */

la acción queda completamente definida, se sabe qué hace y ya se puede interpretar correctamente la secuencia de instrucciones anterior. La especificación podría completarse con la definición,

`Intercambiar x e y` $\equiv$

```
/* x == X && y == Y */
aux = x ;
x = y ;
y = aux ;
/* x == Y && y == X */
```

y, entonces, se sabe también cómo se desarrolla la acción. Este aspecto, sin embargo, muchas veces será secundario para quien simplemente pretenda utilizar la acción *Intercambiar*.

Hasta el momento, la única acción descrita en el entorno de trabajo de la programación era la asignación. Se ha visto cómo utilizarla en secuencias, en condicionales y en bucles. Se ha dicho que estas eran las únicas estructuras de control necesarias para desarrollar programas. El objetivo de este tema es presentar otra herramienta, que si bien no es imprescindible, es de gran utilidad para el diseño de los algoritmos y programas de forma que su estructura sea clara y su mantenimiento sea más sencillo, especialmente cuando son de gran tamaño.

El ejemplo presentado intenta poner de manifiesto las ventajas de esta herramienta. Al utilizar la acción *Intercambiar* se consigue que el diseño lógico del problema de ordenar tres números sea

más fácil de interpretar (es más fácil leer la frase *Intercambiar med y max* que interpretar la correspondiente secuencia de tres asignaciones) y de diseñar (primero se diseña la estructura lógica del problema, y se puede dejar como algo secundario la definición del intercambio).

Además, se realiza menos trabajo. Se escribe menos, ya que se define una vez y se utiliza tres veces... y de paso, se eliminan posibilidades de equivocarse. Y aún cuando haya un error definiendo el proceso del intercambio, sólo hay que corregirlo en un único sitio. Hay que destacar, además, que se ha hecho una definición genérica, sobre `x` e `y`, que luego se usa con valores concretos, `max`, `med` y `min`.

4.1.1. Acciones No Primitivas.

El ejemplo anterior pretende mostrar que, muchas veces, los problemas son más fáciles de expresar y de resolver si se dispone de más acciones que la única definida hasta el momento, la acción primitiva, la asignación. De hecho, los lenguajes de programación brindan la posibilidad de definir acciones propias y de utilizarlas. Son las *acciones no primitivas*.

Definición 4.1 (Acción No Primitiva) *(sub)Programa que resuelve un (sub)problema y que ayuda a enriquecer y ampliar el repertorio de acciones disponibles en un lenguaje.*

Las principales ventajas que se derivan del uso de las acciones no primitivas son las siguientes:

- **Los algoritmos son más inteligibles**.

  Esta herramienta permite formar capas de abstracción; de alguna forma es similar a la obtenida al utilizar un lenguaje de programación en lugar de utilizar el ensamblador, que, a su vez, es más inteligible que el lenguaje máquina. Pero es aún más completa y más útil: un lenguaje de programación tiene un número de acciones limitadas. Y no hay límite en la definición de acciones no primitivas.

  Con esta herramienta se pueden expresar acciones de una forma más natural, más cercana al lenguaje hablado; evidentemente, con limitaciones, puesto que se sigue dependiendo de normas sintácticas y se mantiene la restricción de eliminar las ambigüedades propias del lenguaje natural.

- **Los algoritmos son más fáciles de diseñar**.

  El diseño se puede realizar de forma global, atendiendo a la estructura del problema, sin reparar en pequeños detalles que hagan al programador perder la perspectiva general.

  Una vez esbozada la solución, se puede analizar cada acción no primitiva y resolverla por separado, incluso por diferentes personas. Con la ventaja añadida de que el problema será más pequeño y eso suele traducirse en un algoritmo más simple.

- **Permiten ahorrar trabajo**.

  Un problema se estudia y se resuelve una vez, pero puede utilizarse las veces que se quiera. Con la ventaja añadida de que hay menos posibilidades de equivocarse y de que, si hay un error, sólo hay que corregirlo en la definición de la acción no primitiva.

  Es decir, se puede reutilizar. Y no sólo el trabajo propio, sino que mediante el uso de *bibliotecas de programas*, es decir, colecciones de programas ya diseñados y verificados, es posible utilizar acciones que otros han diseñado. Este hecho es muy habitual. Casi todos los lenguajes disponen de acciones no primitivas de uso muy común en sus *bibliotecas estándar*. Por

ejemplo, en C se dispone de `printf()`, `scanf()`, `fopen()`, `fclose()`,... en la biblioteca estándar de entrada/salida (`stdio`); además, en `math` se dispone de `sqrt()`, `pow()`, `sin()`, `cos()`, `exp()` ... o en `stdlib`, que es la biblioteca de utilidades de uso general, de operaciones como `atoi()`, `rand()`, `srand()`, `malloc()`, `free()`,...

También existen bibliotecas más específicas, destinadas a facilitar el desarrollo de aplicaciones concretas. Un ejemplo clásico (y uno de los más antiguos) es la biblioteca BLAS, *Basic Linear Algebra Subroutines*, que define operaciones básicas para tratar vectores y matrices. Existen bibliotecas gráficas, como OpenGL o todo el sistema X. Y hay bibliotecas de desarrollo, como la STL de C++.

La definición de acciones no primitivas depende del lenguaje que se esté utilizando. En general, se puede decir que tienen una definición similar a la de un programa; es más, tienen su propio entorno de trabajo (sus propias variables) y su propio cuerpo de ejecución.

La diferencia más importante se refiere a cómo reciben los datos y a cómo devuelven los resultados. Volviendo al ejemplo con el que se empezó el tema, la cuestión es: se ha definido la acción *Intercambiar x e y*, ¿cómo hay que hacer para indicar que `x` será unas veces `med` y otras `min` y que `y` será unas veces `max` y otras `med`?

Para poder utilizar acciones no primitivas, se necesita saber utilizar el *mecanismo de comunicación* entre ella y el programa, los *parámetros*.

4.1.2. Concepto de Parámetro.

Hasta el momento, en el entorno de trabajo sólo se había hecho mención de objetos de valor variable (o variables) y de objetos de valor constante (o constantes). Para usar acciones no primitivas hay que introducir un nuevo tipo de objeto, los parámetros, que son necesarios para intercambiar información entre el entorno del algoritmo que invoca a una acción no primitiva y el entorno propio de la acción no primitiva.

Definición 4.2 (Parámetros Formales) *Objetos genéricos que se utilizan para formular una acción. Esa acción no se podrá ejecutar hasta que no se asignen valores concretos a dichos parámetros.*

En el ejemplo, la acción *Intercambiar* se definía en función de dos objetos genéricos, `x` e `y`. Por lo tanto, `x` e `y` son los parámetros formales de *Intercambiar*.

Son imprescindibles para definir la acción no primitiva. La sintaxis concreta depende del lenguaje concreto, pero la definición de acciones no primitivas siempre incluye una *cabecera*, en la que se indica cuál es el *identificador* asociado a dicha acción, y la *lista de parámetros formales*.

Cuando desde un programa se desee utilizar esa acción primitiva, es decir, efectuar una *llamada*, se hará mediante el uso del identificador y la lista de *parámetros reales*.

Definición 4.3 (Parámetros Reales) *(o Actuales) Objetos con valores concretos sobre los que trabaja una acción no primitiva.*

Entre la lista de parámetros formales y la lista de parámetros reales se establece una correspon-

dencia uno a uno, definida por el orden en que aparecen. En el establecimiento de la correspondencia se *debe respetar el tipo*. El nombre no es relevante.

Es posible establecer otra clasificación de los parámetros desde el punto de vista funcional, ya que no todos los parámetros realizan la misma función: en la llamada se han de proporcionar *datos* a la acción no primitiva para que produzca *resultados*. Por ello se distingue entre:

Parámetros de ENTRADA: Son aquellos cuyo valor es tomado como un DATO por la acción no primitiva; es decir, su valor es utilizado como un dato necesario para producir los resultados. Y su valor NO CAMBIA tras la ejecución de la acción no primitiva.

Parámetros de SALIDA: Son aquellos cuyo valor final se interpreta como un RESULTADO de la acción no primitiva; es decir, su valor inicial no interesa, pero su valor final recoge un resultado de las acciones realizadas. Su valor necesariamente CAMBIA al ejecutar la acción no primitiva.

Parámetros de ENTRADA/SALIDA: Son aquellos parámetros cuyo valor inicial se utiliza en la acción no primitiva y que, necesariamente, deben cambiar durante la ejecución.

El mecanismo según el cual se desarrolla esta comunicación varía según el lenguaje de programación, pero se puede hacer una especie de paralelismo con una asignación,

<var> = <expresión>.

Al realizar la llamada, los *parámetros reales de entrada*, se usan para "asignar" valores a los correspondientes parámetros formales. Es decir, un parámetro real de entrada da un valor, a partir del cual trabajar, al correspondiente parámetro formal. Normalmente, este valor se puede dar como una <expresión>, es decir, un valor constante, un nombre de variable o el valor resultante de evaluar una expresión.

A la hora de devolver los resultados, los *parámetros reales de salida* jugarían el papel de <var> y el valor que se les asigna, <expresión>, es el mismo con el que acaban la ejecución los *parámetros formales de salida*. Por lo tanto, un parámetro real de salida, o de entrada/salida, debe ser, necesariamente, una <variable>, ya que debe recoger el resultado que devuelve el correspondiente parámetro formal.

Para acabar de describir el mecanismo, falta comentar qué ocurre cuando, durante la ejecución de un programa, se debe ejecutar una llamada a una acción no primitiva. El procesador no puede ejecutar dos programas al mismo tiempo; por lo tanto, suspende temporalmente la ejecución del programa desde el cual se efectúa la llamada. Esto supone guardar información sobre el punto en que se suspende la ejecución, el valor de las variables, el valor de los parámetros de entrada y el valor de registros de la CPU imprescindibles para restaurar más tarde el estado del entorno, entre otras cosas. Una vez se ha guardado esta información, comienza la ejecución de la acción no primitiva. Cuando finaliza, la CPU reinstaura el entorno del programa y toma nota de los resultados proporcionados por medio de los parámetros de salida, continuando con la ejecución en donde la había dejado.

Bien usada, la parametrización permite asegurar la independencia entre los entornos de trabajo de la acción primitiva y el programa que la utiliza, lo que redunda en una mayor generalidad y posibilita su reutilización. Existe una mala costumbre en programación que es el uso de variables globales. Es una mala costumbre en tanto que resta (y mucho) generalidad al diseño de acciones no primitivas; por lo tanto, se les estaría privando de su mayor potencia. Usar variables globales supone introducir de forma artificial en el entorno de la acción primitiva una variable del programa que la llama. En consecuencia, esa acción sólo podrá ser usada por dicho programa. Entonces, ¿para qué tomarse la molestia de definirla si sólo se puede utilizar ahí?. ¿Qué ocurriría si, por ejemplo, en C hubiera que

redefinir `scanf` cada vez que se quisiera usar? ¿o si sólo se pudieran calcular raíces cuadradas de variables que se llamen `x`?

Funciones y Procedimientos.

Hay lenguajes que clasifican sus acciones no primitivas en función de cómo se realiza la llamada. Pascal (y antes FORTRAN, pero menos formalmente) introdujo esta clasificación.

Según ese lenguaje, una acción no primitiva toma la forma de procedimiento, *procedure*. La llamada se hace como una *instrucción independiente*, mediante el identificador y la lista de parámetros reales.

Pascal distingue, sin embargo, un tipo especial de acción no primitiva, la denominada función, *function*, que se asimila a una función matemática. Por ejemplo, dada la función,

$$f(c) = c^3 + 2c^2 + 4$$

su valor se calcula en función de `c` y es único para cada valor,

$$f(1) = 7, f(2) = 20, f(3) = 49, f(4) = 100, ...$$

La función se caracteriza por devolver un único valor. En Pascal ese valor se asigna directamente sobre el nombre de la función, que actúa como una variable más; otros lenguajes lo hacen asociando ese valor a un parámetro que, mediante una instrucción denominada `return`, lo asocia al nombre de la función.

Sea cual sea el mecanismo, se obtiene "algo" con,

- un **nombre** (el de la función),
- un **valor** (el que devuelve la función),
- un **tipo** (el del valor que devuelve la función),

En consecuencia, *la función puede utilizarse dentro de una expresión*.

Esta distinción entre procedimiento y función, de alguna forma se ha asimilado en otros lenguajes de programación. En concreto, en `C` y en `python`, el formato de una acción no primitiva se puede asimilar al de una función. Pero, de hecho, se verá que implícita o explícitamente su comportamiento se corresponderá con el de un procedimiento o el de una función, según cómo se utilice (cómo se realiza la llamada), el número de valores que devuelva y el mecanismo mediante el cual se devuelven.

4.1.3. Acciones no Primitivas en C.

El mecanismo de definición de acciones no primitivas y de parametrización es muy dependiente del lenguaje. En esta subsección se presenta el mecanismo concreto del lenguaje `C`. Se mostrarán diferentes ejemplos de su definición y su uso. Es decir, de la sintaxis necesaria para definirlas y de la sintaxis necesaria para su uso correcto.

En la definición se debe indicar, en la *cabecera*, el nombre de la acción no primitiva, así como la lista de parámetros necesarios para establecer la comunicación.

En el caso de funciones, la sintaxis es:

```
<tipoDeDatos> <nombreFunción>(lista de parámetros formales) {
  /* Declaración de variables internas */

  /* Bloque de instrucciones */
  <instrucción1>;
  <instrucción2>;
  ...
  <instrucciónN>;

  return <expresión>;
}
```

La primera línea, es la cabecera y en ella se indica:

- `tipoDeDatos`, el tipo del dato que va a devolver la función, que puede ser `int`, `float`, `char`, o cualquier otro tipo definido previamente.
- `nombreFunción`, el nombre que identifica a la función.
- `lista de parámetros formales`, que en el caso de una función también suelen denominarse *argumentos* y que permiten especificar el comportamiento de la función. Tiene la forma `tipo1 id1, tipo2 id2, ..., tipoN idN`. Es decir, por cada parámetro se indica su nombre y su tipo.

  Todos los parámetros formales serán *parámetros de entrada*.

El bloque de instrucciones entre llaves define las operaciones a realizar en la función, utilizando los parámetros formales. La última instrucción de la función ha de ser[1]

```
return <expresión>;
```

e indica el valor que devuelve la función. El resultado de evaluar la `<expresión>` tiene que coincidir en tipo con el `tipoDeDatos` definido en la cabecera de la función.

Por lo que se refiere a los procedimientos, la sintaxis que se utilizará es la siguiente:

```
void <nombreProcedimiento>(lista de parámetros formales) {
  /* Declaración de variables internas */

  /* Bloque de instrucciones */
  <instrucción1>;
  <instrucción2>;
  ...
  <instrucciónN>;
}
```

Como se puede observar es muy similar a la sintaxis utilizada para las funciones. De hecho, los procedimientos en `C` se definen igual que las funciones, pero indicando como tipo de datos a

[1] Aunque no es estrictamente necesario en `C`, en la asignatura siempre se hará así, siguiendo un estilo de programación.

devolver el tipo `void`, que es el tipo "vacío" de `C`. Es decir, se indica que se trata de una función que no devuelve ningún valor asociado al nombre de la función. Por ese motivo, en un procedimiento no existe la instrucción `return` de las funciones. En su lugar, en la lista de parámetros formales se indicará cuáles son parámetros de entrada y cuáles son *parámetros de entrada/salida*[2].

Esto en cuanto a la sintaxis. En cuanto al uso, depende de dónde se haya hecho la definición, que en `C` se puede hacer en el mismo fichero en el que se define el programa que las utiliza, o en un fichero distinto. Esta última esta última opción es más conveniente cuando se trata de acciones no primitivas de uso muy general.

Sea cual sea el fichero donde se ha definido la acción no primitiva, se debe realizar un *prototipo* de la misma. Los *prototipos* permiten utilizar las funciones o procedimientos antes de diseñarlas completamente. El prototipo de una función o procedimiento es su cabecera acabada en "`;`".

<**tipoDeDatos**> <**nombreFunción**>**(lista de parámetros formales);**

Obviamente, esto no evita el tener que diseñarla y escribirla completamente en algún momento. Sólo sirve para poder utilizar la función (o procedimiento) sin necesidad de tener que diseñarla en este momento (por ejemplo, porque la está diseñando otra persona).

Si la definición se hace en el fichero del programa que las usa, se debe hacer la declaración de prototipos antes de definir la función principal; y es una práctica recomendable que la definición de las acciones no primitivas se haga *después* de la definición de la función principal.

Si la definición se hace en un fichero distinto al del programa que utilizará la acción no primitiva, además del propio fichero con la definición, se debe realizar un *fichero de cabeceras* que incluya únicamente, los prototipos de las acciones no primitivas. El fichero de definición debe tener la extensión `.c`, mientras que el fichero de cabeceras debe tener la extensión `.h`. Una vez que se dispone del fichero de cabeceras, con la declaración de prototipos, para utilizar las funciones y procedimientos se debe usar la directiva `#include "nombreFichero.h"`; el efecto es el mismo que cuando se utiliza con ficheros de la biblioteca estándar del lenguaje.

De acuerdo a lo anterior, la estructura de un programa en `C` se amplía al siguiente esquema:

```
/* Ficheros de Cabecera */

/* Declaración de constantes */

/* Declaración de prototipos */

/* Función principal*/

int main() {
   /* Declaración de variables de main() */

   /* Cuerpo del algoritmo */
}

/* Definición del resto de funciones y procedimientos */
```

[2]El lenguaje `C` no dispone de parámetros de salida puros.

El ejemplo de la subsección 10.1.2 ilustra mejor por qué unas veces conviene definir funciones y procedimientos en el propio fichero, y otras, en uno independiente; también amplia los detalles de cómo hacerlo. Por ahora, el objetivo es distinguir entre entre funciones y procedimientos y conocer cómo se utilizan. Por eso, a continuación, se verán varios ejemplos de definición y uso. En ellos, se asume que las implementaciones se han realizado en un fichero denominado `ejemplosT4.c`, cuyos prototipos se definen en el fichero `ejemplosT4.h`.

Funciones: Un único parámetro de salida.

En este caso, sólo se devuelve un valor asociado al nombre de la acción no primitiva mediante `return`. Para llamar a la acción se usará el número adecuado de parámetros reales de entrada; el valor del resultado se identifica con el nombre de la función. Por lo tanto, se puede usar la llamada en cualquier sitio donde se pueda usar una variable.

Sólo se utilizan parámetros de entrada; el valor de los parámetros reales se copia sobre los correspondientes parámetros formales y la función se ejecuta tomando dichos valores como iniciales. El único valor que se comunica al exterior es el devuelto mediante `return`. Es decir, el valor de los parámetros reales no cambia, ya que la función trabaja sobre una copia de los mismos.

- Cálculo del factorial de `n`.

  Para calcular el valor asociado al factorial de `n`, hay que acumular el producto de todos los enteros entre `1` y `n`.

```
int fact(int n){
/* pre: n=N, entero positivo */

   int f,i;

   f=1;
   for (i=1; i<=n, i++){
      f=f*i;
   }

   return f;
}
/* post: f=N!, entero positivo */
```

  A esta función le corresponde el prototipo,

```
int fact(int n);
/* pre: n=N, entero positivo */
/* post: f=N!, entero positivo */
```

  Como ejemplo de uso, se propone el cálculo de las combinaciones de `m` elementos tomados de `n` en `n`, que se calcula como

$$C_n^m = \frac{m!}{n! * (m-n)!} .$$

combinaciones.c
```
#include <stdio.h>
#include "ejemplosT4.h"

int main(){
  /* Datos */
  int m,n;
  /* Resultado */
  int comb;

  printf("Valor de n: ");
  scanf("%d", &n);
  printf("\nValor de m: ");
  scanf("%d", &m);
  comb=fact(m)/(fact(n)*fact(m-n));
  printf("\n\nCombinaciones de %d, de %d en %d, son %d",m,n,n,comb);
}
```

- Determinar si un valor entero `n` es primo.

  Un número es primo si no tiene más divisores que el `1` o él mismo. Para comprobar que un número es primo es suficiente con comprobar que los demás candidatos a divisores, los números comprendidos entre `2` y `n/2`, no lo son. Además, en el momento en que se encuentre un divisor se detiene el proceso (si hay un divisor, no es primo).

```
boole primo(int n){
/* pre: n=N, entero positivo */

   int i;
   boole esPrimo;

   esPrimo=cierto;
   i=2;
   while ((i <= n/2) && (esPrimo)){
      esPrimo=n%i != 0;
      i++;
   }

   return esPrimo;
}
/* post: falso(0), si n no es primo; cierto, si n es primo */
```

  Prototipo,

```
boole primo(int n);
/* pre: n=N, entero positivo */
/* post: falso si n no es primo; cierto, si n es primo */
```

  Si se devuelve `falso`, `n` no es primo; si devuelve `cierto`, `n` es primo. Se puede utilizar para escribir la tabla de todos los números primos entre `1` y `k`,

tablaPrimos.c
```
#include <stdio.h>
#include "ejemplosT4.h"
```

```
int main(){
  /* Datos */
  int k;
  /* Resultado: se visualiza */

  int i;/* Variable de main() */
        /* NO tiene NINGUNA relación con la definida en primo */

  printf("Valor de k: ");
  scanf("%d", &k);

  for (i=1; i<=k; i++){
      if (primo(i)){
          printf("\t%d", i);
      }
  }
}
```

Cuestión: ¿Dónde debería estar la definición del tipo `boole`?

- Suma de divisores propios de un entero, n.

  Sabiendo que el `1` es un divisor, se inicializa a ese valor un acumulador llamado `suma`; se comprueba si los demás candidatos, enteros del `2` al `n/2`, lo son o no. Cada vez que se encuentra un divisor se acumula sobre `suma`.

```
int sumaDiv(int n){
/* pre: n=N, N entero, N>1 */

   int i, suma;

   suma=1;
   for (i=2; i<=n/2 ; i++){
      if (n%i == 0){
          suma=suma+i;
      }
   }

   return suma;
}
/* post: suma, entero, contiene la suma de divisores propios de n */
```

  Prototipo,

```
int sumaDiv(int n);
/* pre: n=N, entero positivo */
/* post: suma, entero, contiene la suma de divisores propios de n */
```

  Se puede utilizar para escribir otra acción que determina si dos números son o no son amigos. Sabiendo sumar los divisores propios, basta con transcribir la definición: n1 y n2 son amigos si n1 es igual a la suma de divisores propios de n2, y n2 es igual a la suma de divisores propios de n1, lo que se puede comprobar así:

```
boole amigos(int n1, int n2){
/* pre: n1=N1, n2=N2, enteros positivos */

   int aux;
   boole loSon;

   aux=sumaDiv(n1);
   if (aux == n2){
      loSon = (n1 == sumaDiv(n2))
   }
   else {
      loSon = falso;
   }
   return loSon;
}
/* post: loSon vale cierto si N1 y N2 son amigos, */
/* falso en caso contrario */
```

o así:

```
boole amigos(int n1, int n2){
/* pre: n1=N1, n2=N2, enteros positivos */

   boole loSon;

   loSon = falso;
   if (sumaDiv(n1) == n2){
      loSon = (n1 == sumaDiv(n2))
   }
   return loSon;
}
/* post: loSon vale cierto si N1 y N2 son amigos, */
/* falso en caso contrario */
```

Cuestión: Tal y como se ha utilizado `sumadiv` al construir `amigos`, ¿en qué fichero se habrá definido la función `amigos`?

- Máximo Común Divisor de dos enteros, u y v.

  Se puede definir mediante el algoritmo de Euclides, que es el algoritmo más antiguo que se conoce (se estima que fue formulado en el año 300 A.C.) :

```
int maxComDiv(int u, int v){
/* pre: u=U,v=V, enteros positivos */

   while (u != v) {
      if (u > v){
          u = u - v;
      }
      else {
          u = v - u;
      }
   }
```

```
   return u;
}
/* post: devuelve el máximo común divisor de U y V */
```

Prototipo,

```
int maxComDiv(int u, int v);
/* pre: u=U,v=V, enteros positivos */
/* post: devuelve el máximo común divisor de U y V */
```

y, se puede utilizar, por ejemplo, para comprobar si dos números son primos entre sí; si lo son, su máximo común divisor es `1`:

primosEntreSi.c

```
#include <stdio.h>
#include "ejemplosT4.h"

int main(){
  /* Datos */
  int x1,x2;
  /* Resultado: se visualiza */

  printf("Valor del primer número: ");
  scanf("%d", &x1);
  printf("\nValor del segundo número: ");
  scanf("%d", &x2);

  if (maxComDiv(x1,x2)==1){
      printf("\nSon primos entre sí.");
  }
  else {
      printf("\nNo son primos entre sí.");
  }
}
```

Cuestión: ¿Con qué valor acaba x1 tras la ejecución de la llamada a `maxComDiv`?

Procedimientos: ninguno, uno o más de un parámetro de salida.

En este caso, la llamada es una instrucción independiente y *no se puede utilizar* en una expresión; se puede devolver más de un valor (o ninguno o uno), y la comunicación que el lenguaje `C` establece es por medio de parámetros de entrada/salida. Por lo tanto, en un procedimiento se podrán tener parámetros de entrada (y la comunicación se establece igual que en el caso de las funciones) y parámetros de entrada/salida.

En el caso de los parámetros de entrada/salida, (E/S), el valor de los parámetros reales se copia sobre los formales, el procedimiento realiza operaciones con dichos valores y devuelve los resultados en los parámetros formales, que sobreescriben el valor de sus correspondientes parámetros reales.

En `C` los parámetros formales de E/S se escriben anteponiendo un `*` al nombre en la definición de parámetros, y *siempre que se utilicen* en el cuerpo de instrucciones del procedimiento. En la lista de parámetros reales, los que sean de E/S, se escriben anteponiendo el símbolo `&` a su nombre. Los parámetros reales de E/S sólo pueden ser variables.

- Cálculo del cociente y el resto de la división entera de A y B.

```
void divEnt(int dvdo, int dvsor, int *coc, int *resto){
/* pre: dvdo=A, dvsor=B, enteros A>=0, B>0 */

   *coc = 0;
   *resto = dvdo;
   while (*resto >= dvsor){
      *resto = *resto - dvsor;
      *coc = *coc + 1;
   }
}
/* post: coc = A/B, entero y resto=A%B, entero*/
```

Prototipo:

```
void divEnt(int dvdo, int dvsor, int *coc, int *resto);
/* pre: dvdo=A, dvsor=B, enteros A>=0, B>0 */
/* post: coc = A/B, entero y resto=A%B, entero*/
```

Se podría utilizar, por ejemplo, de la siguiente forma, entre las instrucciones de cualquier programa:

ejDivEnt.c

```
#include <stdio.h>
#include "ejemplosT4.h"

int main(){
  /* Datos */
  ....
  /* Resultados */
  ....
  /* Declaración de variables */
  ....
  int n1, n2, m1, m2;
  ....

  /* Cuerpo de instrucciones */
  .....
  divEnt(n1, n2, &m1, &m2);
  /*ahora, el valor de m1 es (n1/n2) y el de m2 es (n1%n2)*/
  ....
}
```

- Una acción no primitiva que calcule el máximo común divisor y el mínimo común múltiplo de dos enteros. Ya se sabe calcular el mcd; el mcm se calcula como el producto de los dos enteros, dividido por el mcd.

```
void mcdMcm(int a, int b, int *mcd, int *mcm){
/* pre: a=A, b=B, enteros positivos */

   *mcd = maxComDiv(a,b);
   *mcm = (a * b)/*mcd;
}
/* post: mcd=mcd(A,B) y mcm=mcm(A,B) */
```

Prototipo:

```
void mcdMcm(int a, int b, int *mcd, int *mcm);
/* pre: a=A, b=B, enteros positivos */
/* post: mcd=mcd(A,B) y mcm=mcm(A,B) */
```

Para usarla:

mcdYMcm.c

```
#include <stdio.h>
#include "ejemplosT4.h"

int main(){
  /* Datos */
  int xx,yy;
  /* Resultado */
  int mcd, mcm;

  printf("Valor del primer número: ");
  scanf("%d", &xx);
  printf("\nValor del segundo número: ");
  scanf("%d", &yy);

  mcdMcm(xx, yy, &mcd, &mcm);

  printf("\nEl Mcd de %d y %d es %d.", xx, yy, mcd);
  printf("\nEl Mcm de %d y %d es %d.", xx, yy, mcm);
}
```

- Procedimiento para leer el valor de tres ángulos, verificando además que suman 180 grados:

```
void lecturaCorrecta(float *a1, float *a2, float *a3){
/* pre: no hay datos, se leen a1, a2 y a3 de teclado */

   float aux1, aux2, aux3;

   printf("Lectura de tres ángulos.\n");
   printf("Sus valores no pueden sumar más de 180 grados.\")
   do {
      printf("\tPrimer ángulo: ");
      scanf("%f", &aux1);
      printf("\n\tSegundo ángulo: ");
      scanf("%f", &aux2);
      printf("\n\tTercer ángulo: ");
      scanf("%f", &aux3);
   } while ((aux1+aux2+aux3) != 180.0);

   *a1 = aux1;
   *a2 = aux2;
   *a3 = aux3;
}
/* post: a1, a2, a3, reales y a1 + a2 + a3 = 180 */
```

Prototipo:

```
void lecturaCorrecta(float *a1, float *a2, float *a3);
/* pre: no hay datos, se leen a1, a2 y a3 de teclado */
/* post: a1, a2, a3, reales y a1 + a2 + a3 = 180 */
```

En el siguiente ejemplo, se verá cómo usarlo, al combinarlo con otra acción no primitiva que clasifica un triángulo por el valor de sus ángulos.

- Clasificación de un triángulo según el valor de sus ángulos: hay que ver si hay alguno recto, y entonces es rectángulo, si todos son agudos, y entonces es acutángulo. En otro caso, es obtusángulo.

  En este procedimiento no se devuelven resultados: sería un ejemplo de procedimiento con 0 parámetros de E/S.

```
void clasificacion(float a1, float a2, float a3){
/* pre: a1, a2, a3, reales y a1 + a2 + a3 = 180 */

   if ((a1 == 90.0) || (a2 == 90.0) || (a3 == 90.0)){
       printf("El triángulo es rectángulo.");
   }
   else {
        if ((a1 < 90.0) && (a2 < 90.0) && (a3 < 90.0)){
             printf("El triángulo es acutángulo.");
        }
        else {
             printf("El triángulo es obtusángulo.");
        }
   }
}
/* post: no hay resultados, se visualiza en pantalla */
```

Prototipo:

```
void clasificacion(float a1, float a2, float a3);
/* pre: a1, a2, a3, reales y a1 + a2 + a3 = 180 */
/* post: no hay resultados, se visualiza en pantalla */
```

Para usarlo, se puede combinar con la acción no primitiva que lee tres ángulos comprobando que su suma es igual a 180 grados.

clasifTriang.c

```
#include <stdio.h>
#include "ejemplosT4.h"

int main(){
  /* Datos */
  float angulo1, angulo2, angulo3;
  /* Resultado por pantalla*/

  printf("Introduzca el valor de tres ángulos y
                                clasificaré el triángulo.");
  printf("\n-----------------\n\n");

  lecturaCorrecta(&angulo1, &angulo2, &angulo3);
  clasificacion(angulo1, angulo2, angulo3);
}
```

- A veces, viene bien disponer de una acción que se limite a escribir una mensaje en pantalla, por ejemplo, para informar a un usuario de cómo debe usar un programa. No se necesita ningún parámetro de entrada ni de salida.

```
void usage(){
/* pre: no hay datos */

   printf("Uso de este programa:");
   printf("\n\t(1) Introduzca las medidas de los dos segmentos.");
   printf("\n\t(2) Introduzca un valor infinitesimal (epsilón).");
   printf("\n\t(3) Para finalizar, introduzca tres ceros.");
}
/* post: no hay resultados, sólo visualiza cómo operar */
```

Prototipo:

```
void usage();
/* pre: no hay datos */
/* post: no hay resultados, sólo visualiza cómo operar */
```

4.2. Especificación de Acciones no Primitivas.

Como cualquier acción, una acción no primitiva también admite una especificación, con una precondición y una postcondición.

Para definir la especificación de la acción primitiva, la asignación, se presentó el siguiente esquema,

/* <expresion> *satisface precondición* */
<var> = <expresion>
/* <var> *satisface postcondición* */

Para las acciones no primitivas se establece un mecanismo similar, pero en relación a la parametrización. Así, para una función se define como,

/* <parámetros de entrada> *satisfacen precondición* */
<var> = <identificador>(<parámetros de entrada>)
/* <var> *satisface postcondición* */

y, para un procedimiento,

/* <parámetros de entrada> *satisfacen precondición* */
<identificador>(<parámetros de entrada>, <parámetros de salida>)
/* <parámetros de salida> *satisfacen postcondición* */

Es decir, en general se establecen qué condiciones deben satisfacer los parámetros de entrada y qué condiciones deben satisfacer los parámetros de salida. De esta forma, además de saber qué hace la acción no primitiva, se estará en condiciones de asegurar cuándo funcionará correctamente.

En los ejemplos anteriores, se ha estado realizando la especificación de forma intuitiva. Un ejemplo formal, sería el siguiente, en el que se especifica la acción no primitiva `divEnt`:

/* (a == $\mathcal{A}$) && (b == $\mathcal{B}$) && ($\mathcal{A} \geq 0$) && ($\mathcal{B} > 0$) */
`divEnt`(a, b, &q, &r)
/* ($\mathcal{A}$ == $\mathcal{B}$ * q + r) && ($0 \leq$ r $< \mathcal{B}$) */

Esta notación se puede generalizar al programa entero, identificando previamente los datos y los resultados[3]. De alguna forma, la verificación formal de un programa pasa por satisfacer su precondición y su postcondición. Se puede dividir el encadenamiento lógico de postcondiciones y precondiciones en pasos intermedios, aplicando a cada estructura de control los mecanismos establecidos en el tema anterior, pero el efecto ha de ser la derivación de la postcondición final de la precondición inicial.

4.2.1. Dominio de Definición.

En relación con la especificación de un algoritmo, surge el concepto de las *instancias permitidas*. En cualquier algoritmo, se deben identificar los datos, con los que se va a desarrollar el proceso para obtener los resultados.

De alguna forma, ya sea explícita (porque el algoritmo se definirá como una acción no primitiva), ya sea implícita (porque, aunque se esté definiendo un programa y no se esté obligado a enunciar una lista de parámetros, se debe saber con qué datos va a trabajar y qué resultados va a obtener), siempre que se define un proceso por medio de un algoritmo se debe tener en cuenta cuáles serán sus parámetros formales.

En una lista de parámetros se indican nombres, se puede indicar el tipo, pero no se está obligado a pensar en qué valores son válidos para que el proceso se desarrolle correctamente. Por ejemplo,

```
int unEjemplo(int a, float b, ...){

   /* variables */
   .....
   float j;
   .....

   /* cuerpo */
   .....
   j = b/a;
   .....
}
```

En este ejemplo, se resalta solamente el cálculo `b/a`. El motivo es que el parámetro `a`, en principio, podría ser cualquier valor entero *menos el valor 0*, ya que la división por cero no es una operación definida y se produciría un error.

[3] La definición de las acciones no primitivas obliga a definir parámetros y, por lo tanto, datos y resultados. Hasta este tema, no se estaba obligado a identificar de forma explícita datos y resultados, pero debía hacerse: para que el programa funcionase había que saber qué leer y qué escribir. Esos eran los datos y los resultados. Y sobre ellos se debe estudiar la especificación del programa.

Pero el concepto de instancia permitida no sólo se refiere al ámbito de los posibles errores de ejecución, sino que adquiere su verdadera dimensión cuando se relaciona con el propio comportamiento del algoritmo. Para ilustrar este hecho, se propone como ejemplo el problema de la multiplicación de números enteros.

Si se pretende realizar la operación $19 \times 45 = 855$, por ejemplo, se puede utilizar entre otros el método de *Multiplicación a la Rusa*; en este método se sigue la siguiente idea: se escribe el multiplicador y el multiplicando, y se forman sendas columnas bajo cada uno de ellos de forma que bajo el multiplicador se dividen los números por 2 (división entera) y bajo el multiplicando se multiplican los números por 2. Se procede así hasta que en la columna del multiplicador se llegue al valor 1 para, a continuación, sumar todos aquellos números que estén en la columna del multiplicando asociados a números impares en la columna del multiplicador. Con los valores anteriores, el proceso sería,

45	19	19
22	38	
11	76	+76
5	152	+152
2	304	
1	608	+608
		855

Cada uno de los algoritmos que permiten solucionar este problema tiene ventajas e inconvenientes. Para utilizar este método, por ejemplo, sólo se requiere saber multiplicar y dividir por 2. Pero presenta otra característica que lo hace más interesante para ilustrar el problema de las instancias permitidas: el algoritmo de Multiplicación a la Rusa, *no produce una solución correcta si el multiplicador es negativo*.

De acuerdo a los ejemplos anteriores, se define,

Definición 4.4 (Dominio de Definición) *El conjunto de instancias permitidas; es decir, el conjunto de todos los valores para los que es válido un algoritmo.*

La multiplicación es una función definida sobre $\mathcal{Z} \times \mathcal{Z}$ en $\mathcal{Z}$. El algoritmo de multiplicación, siguiendo el método anterior, tendría dos parámetros de entrada de tipo ENTERO y un parámetro de salida también de tipo ENTERO. Sin embargo, su Dominio de Definición es $\mathcal{Z} \times \mathcal{N} \longrightarrow \mathcal{Z}$, es decir, el multiplicador debe ser positivo. No se producirán errores de ejecución como en el caso de la división por cero; simplemente, el método no produce la solución correcta cuando el multiplicador es negativo.

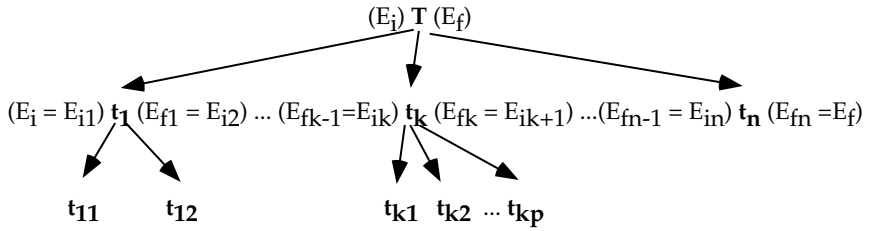
El concepto de dominio de definición permite formular formalmente una distinción importante entre algoritmo y programa: *la implementación supone limitar el dominio de definición de acuerdo a los límites de la propia máquina o del lenguaje de programación*. El algoritmo será válido para su dominio de definición; el programa tiene más limitaciones.

> "Todo ordenador real tiene un límite sobre el tamaño de las instancias de un problema. Este límite nunca puede ser imputado al algoritmo utilizado, lo que muestra una vez más la diferencia esencial entre programas y algoritmos".

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

El uso de acciones no primitivas y su correcta especificación, tiene especial interés en relación a la técnica de *Diseño Descendente* (Top-Down, en nomenclatura inglesa) de algoritmos. La idea básica es la siguiente:

> Dado un trabajo T, descrito por un enunciado no primitivo, que no se puede realizar directamente en un único paso, el análisis descendente de T consiste en encontrar una descomposición $t_1, t_2, \ldots, t_n$ que sea una sucesión de enunciados cuya ejecución realice el trabajo T. Para cada enunciado t_k, hay dos casos posibles:
>
> - t_k es una acción primitiva y se detiene el análisis descendente,
> - t_k no es una acción primitiva y se descompondrá en una sucesión $t_{k1}, t_{k2}, \ldots, t_{kp}$.



El uso de acciones no primitivas permite expresar la descripción del trabajo a realizar siguiendo estas directrices. Entre las ventajas de esta técnica hay que recordar las enunciadas en la primera sección: no sólo se facilita el diseño, sino que además mejora la legilibilidad (siempre y cuando los nombres de las acciones no primitivas sean significativos), lo que redundará en una mejora del mantenimiento del software.

Además, hay que tener en cuenta el siguiente aspecto: el esquema anterior, además de la descomposición del trabajo inicial en tareas más simples y por lo tanto más fáciles de resolver, refleja como cada trabajo t_k pasa el entorno de un estado E_{ik} (estado inicial del trabajo t_k) a un estado E_{fk} (estado final del trabajo t_k).

Por lo tanto, esta técnica enlaza con la idea de la especificación de acciones no primitivas y la división del trabajo: en la medida en que cada t_j ($j \neq 1$) sólo depende del trabajo precedente, t_{j-1}, por el resultado que espera de él ($E_{fj-1} = E_{ij}$ y la postcondición de t_{j-1} coincide con la precondición de t_j), puede enunciarse la descomposición independiente de estos trabajos.

Es decir, la resolución de cada trabajo es independiente; su relación con los otros trabajos se limita al **estado inicial** del entorno (que coincide con el final del trabajo anterior) y al **estado final** del entorno (que coincide con el inicial del trabajo siguiente).

Al hablar de Proyectos Informáticos, esto significa que, en la medida en que el trabajo global tenga una descomposición lógica, modificar o escribir cada una de estas tareas es independiente del resto, con lo que se simplifica la escritura y mantenimiento del software y el trabajo en equipo de forma eficiente. Otro aspecto positivo, y ya comentado, se refiere a la posibilidad de reutilizar los algoritmos, de forma que se creen bibliotecas de programas que eliminan la necesidad de reescribir código: si un determinado subproblema ya está resuelto, simplemente habrá que utilizar el programa ya diseñado.

Todo esto es cierto siempre y cuando se respeten las reglas del juego y no se modifique el análisis inicial del problema, una vez que se haya completado. Muchas veces, el analista y el programador son la misma persona y se cae en la tentación de no respetar ni los pasos enumerados en el primer tema para el diseño de aplicaciones y sistemas informáticos, ni el análisis realizado. Es decir, por facilitar "momentáneamente" el desarrollo de uno de los trabajos intermedios, se debilitan sus precondiciones o sus postcondiciones, se "asumen" propiedades de los datos que le serán entregados para procesar desde la etapa anterior o los que debe entregar a la posterior y... y no sólo se hace una vez, sino dos, tres, las que haga falta. El resultado: tanto el análisis del problema como la descomposición lógica del trabajo se ven afectados. En esas condiciones, verificar el trabajo realizado se hace díficil y asegurar su correcto funcionamiento casi imposible.

4.3.1. Un Ejemplo en C.

Se quiere diseñar un programa que permita a un usuario realizar un proceso geométrico básico, con figuras planas. Tendrá cuatro opciones:

1. Calcular el área de un círculo,
2. Calcular el área de un cuadrado,
3. Calcular el área de un restángulo,
4. Determinar la longitud de un segmento.

El proceso completo se puede describir a grandes rasgos de la siguiente manera:

- Pedir al usuario una opción,
- En función de la opción, realizar el proceso,
- Informar al usuario del resultado obtenido.

¿Qué supone cada uno de estos pasos?

Pedir opción Para que el usuario pueda elegir una opción, parece necesario que sepa de cuáles dispone para, posteriormente, dejarle optar. Se puede pensar en dos acciones:

Informar al usuario Simplemente debe consistir en presentarle un menú de opciones por pantalla. Es una acción sin datos y sin resultados, no es necesario especificarla.

Leer la opción En este caso, sí se produce un resultado, la opción elegida por el usuario y no se debería permitir la introducción de un valor erróneo: se impone como postcondición que la opción sea un valor correcto, que permita optar entre una de las cuatro opciones válidas.

Realizar el proceso Con las postcondiciones impuestas a las tareas anteriores se puede establecer la precondición de esta: recibirá como dato un valor válido de opción. Con estas premisas se pasará a desarrollar uno de los cuatro procesos posibles.

Área del círculo Para poder realizarlo, se necesita como dato el radio del círculo; este dato debe ser un valor positivo. Por lo tanto, se impone como precondición que el radio sea un valor real positivo.

Área del cuadrado El dato necesario para este proceso es el lado del cuadrado, que también debe ser un valor positivo; por lo tanto, también en este caso la precondición impuesta será que el lado sea un valor real positivo.

Área del rectángulo En este caso, se necesitan dos datos, las longitudes de los dos lados. Por el mismo motivo que en los procesos anteriores, se impone como postcondición que sean valores reales positivos.

Longitud del segmento Para determinar la longitud de un segmento se deben conocer las coordenadas del punto de inicio y del punto de fin. La única precondición que se impone es que sean valores reales.

Todos estos procesos tienen un postcondición similar: devolverán como resultado un valor real, que será positivo. Esa será la postcondición de la tarea *Realizar Proceso*.

Mostrar Resultado Esta tarea simplemente recibe un dato, el resultado del proceso. Se puede especificar que será un valor real positivo. Se puede completar indicando también cuál es la opción elegida (por supuesto, dentro del rango correcto), para personalizar el mensaje que se ofrece al usuario. No tiene resultados que ofrecer, puesto que se limita a informar por la pantalla de cuál es el valor de este resultado.

Con esto, se han *especificado* las acciones. Hay que pasar al *diseño*.

Pedir Opción Para realizarlo, sólo hay que diseñar cada uno de los pasos elementales en los que se ha descompuesto:

Informar al usuario Se puede diseñar una acción no primitiva que imprima por pantalla la información o se puede optar por presentarla directamente. En este caso, se opta por definir la acción no primitiva, de forma que el menú pueda presentarse al usuario las veces que se considere oportuno (por ejemplo, si se equivoca eligiendo las opciones).

Leer opción Hay que pedir un valor e insistir hasta que se cumpla que ese valor es una opción válida.

Realizar el proceso Debe consistir en una estructura condicional que nos permita elegir entre cuatro procesos en función del valor de la opción. Las tres tareas a elegir son:

Área del círculo Hay que saber el valor del radio del círculo, que debe ser un valor real positivo y, después, aplicar la correspondiente fórmula matemática. Parece oportuno descomponer la acción en **leeRealPositivo** (reutilizable) y el propio cálculo (podría ser reutilizable en el futuro).

Área del cuadrado Un razonamiento similar nos llevaría a descomponer esta acción también: reutilizando **leeRealPositivo** se puede leer el lado del cuadrado y asegurar que se cumple la precondición y, posteriormente, realizar ya el cálculo.

Área del Rectángulo También en esta tarea se puede reutilizar **leeRealPositivo** en la lectura de cada uno de los lados y realizar después el correspondiente cálculo; además, y ya que un cuadrado es un rectángulo con los 4 lados iguales, se puede pensar en reaprovechar una única acción no primitiva para ambos cálculos.

Longitud del segmento Hay que leer las coordenadas de dos puntos; se podría pensar en una acción **leePunto** (reutilizable) y en el cálculo posterior de la longitud.

Cada uno de estos cálculos se puede implementar como una acción no primitiva: es posible que en aplicaciones posteriores puedan ser útiles.

Mostrar Resultado Si se asume que como dato se reciben la opción elegida y el resultado del proceso, consistirá en una estructura condicional que, en función de la opción, emita el mensaje adecuado, además del resultado.

De este análisis, se desprende la necesidad de implementar una serie de acciones no primitvas. Se propone la siguiente:

```
void muestraMenu(){
/* Informa al usuario de sus opciones */
/* Pre: no hay datos */
  printf("Introduzca opción:\n");
  printf("\t1.- Área de un círculo\n");
  printf("\t2.- Área de un cuadrado\n");
  printf("\t3.- Área de un rectángulo\n");
  printf("\t4.- Longitud de un segmento\n");
}
/* Post: no hay resultados */
```

```
int leerOpcion(int min, int max) {
/* Lee un valor entero que se identifica con una opción entre min y max*/
/*Pre: min y man son enteros,acotan el rango de opciones válidas*/
  int opc;

  printf("\n > Opción elegida (entre %d y %d)? ", min, max);
  scanf("%d", &opc);
  while (opc<min || opc>max){
    printf("\n Opción no válida.");
    printf("\n > Opción elegida (entre %d y %d)? ", min, max);
    scanf("%d", &opc);
   }
  return opc;
}
/* Post: el valor de opc es un entero entre min y max */
```

```
int opciones() {
/* Pide opción al usuario: muestra el menú y lee la opción */
/* Pre: no hay datos */
  int opc;

  mostraMenu();
  opc=leerOpcion(1,4);
  return opc;
}
/* Post: el valor de opc es un entero entre 1 y 4 */
```

```
float leeRealPositivo() {
/* Lee de teclado un valor real positivo */
/* Pre: no hay datos */
  float num;

  scanf("%f", &num);
  while (num<0) {
    scanf("%f", &num);
```

```
  }
  return num;
}
/* Post: num es un valor real positivo */
```

```
void leePunto(float *x, float *y) {
/* Lee de teclados las dos coordenadas de un punto */
/* Pre: no hay datos */
  float cx, cy;

  printf("\n Coordenada x: ");
  scanf("%f",&cx);
  printf("\n Coordenada y: ");
  scanf("%f",&cy);
  *x=cx;
  *y=cy;
}
/* Post: x e y son valores reales */
```

```
float calculaAreaCirculo(float radio) {
/* Dado el radio, calcula el área de un círculo */
/* Pre: radio es un valor real positivo */
  float area;

  area=PI*radio*radio;

  return area;
}
/*Post: area es un valor real y coincide con el área del círculo*/
```

```
float areaCirculo() {
/* Pide el valor del radio y calcula el área del círculo */
/* Pre: no hay datos */
  float area;
  float radio;

  printf("\nValor del radio? ");
  radio=leeRealPositivo();
  area=calculaAreaCirculo(radio);

  return area;
}
/* Post: area tiene un valor real que coincide con el */
/* área del círculo cuyo radio es radio */
```

```
float calculaAreaRectangulo(float lado1, float lado2) {
/* Dados los lados, calcula el área de un rectangulo */
/* Pre: lado1 y lado2 son valores reales positivos */
  float area;
```

```
  area=lado1*lado2;

  return area;
}
/*Post: area es un valor real, coincide con el área de la figura*/
```

```
float areaCuadrado() {
/* Pide el valor del lado y calcula el área del cuadrado */
/* Pre: no hay datos */
  float area;
  float lado;

  printf("\nValor del lado?  ");
  lado=leeRealPositivo();
  area=calculaAreaRectangulo(lado,lado);

  return area;
}
/* Post: area tiene un valor real que es el área */
/* del cuadrado cuyo lado es lado */
```

```
float areaRectangulo() {
/* Pide el valor de los lados y calcula el área del rectángulo */
/* Pre : no hay datos */
  float area;
  float lado1,lado2;

  printf("\nValor del primer lado?  ");
  lado1=leeRealPositivo();
  printf("\nValor del segundo lado?  ");
  lado2=leeRealPositivo();
  area=calculaAreaRectangulo(lado1,lado2);

  return area;
}
/* Post: area es real y su valor es el área del rectángulo */
/*  de lados lado1 y lado2 */
```

```
float calculaLongSegmento(float x1, float y1,float x2, float y2) {
/* Calula la longitud del segmento entre (x1,y1) y (x2,y2) */
/* Pre: x1, x2, y1, y2, son reales */
  float long;
  float a1,a2;

  a1=x1-x2;
  a2=y1-y2;

  long=sqrt(a1*a1+a2*a2);
```

```
  return long;
}
/* Post: long contiene un valor real positivo, */
/* la longitud del segmento */
```

```
float longitudSegmento() {
/* Lee 2 puntos y calcula la longitud del segmento que definen */
/* Pre: no hay datos */
  float x1,y1,x2,y2;
  float long;

  printf("\n Valor del primer punto? ");
  leepunto(&x1,&y1);
  printf("\n Valor del segundo  punto? ");
  leepunto(&x2,&y2);
  long=calculaLongSegmento(x1, y1, x2, y2);

  return long;
}
/* Post: long es un real positivo cuyo valor es */
/* la longitud del segmento */
```

```
float ejecutaOpcion(int opc) {
/* Realiza un proceso determinado en función del valor de opc */
/* Pre: opc es un valor entero entre 1 y 4 */
  float res;
  switch (opc)  {
   case 1:
     res=areaCirculo();
     break;

   case 2:
     res=areaCuadrado();
     break;

   case 3:
     res=areaRectangulo();
     break;

   case 4:
     res=longitudSegmento();
     break;
   }
  return res;
}
/* Post: res es un valor real, que coincide con el */
/* resultado del proceso elegido */
```

```
void muestraResultado(int opc, float resultado) {
/* Muestra el resultado del proceso elegido */
```

```
/*Pre:opc es un valor entero entre 1 y 4 y resultado es un real*/
  switch (opc) {
   case 1:
     printf("\nEl Área del círculo es %f.\n", resultado);
     break;

   case 2:
     printf("\nEl Área del cuadrado es %f.\n", resultado);
     break;

   case 3:
     printf("\nEl Área del rectángulo es %f.\n", resultado);
     break;

   case 4:
     printf("\nLa longitud del segmento es %f.\n", resultado);
     break;
   }
}
/* Post: no hay resultados */
```

De alguna forma es como si se dispusiera de las piezas de un rompecabezas y ahora sólo faltara componerlas para obtener el proceso global. Un primer paso puede ser distinguir entre la acciones que son más genéricas y que, por lo tanto, podrían ser reutilizadas en el futuro, y las que son más concretas y en su diseño se ha tenido muy en cuenta el tipo de aplicación que se pretendía resolver.

Esta distinción resulta especialmente indicada si se tiene en cuenta lo que se comentó en la subsección 4.1.3: las acciones no primitivas en C se pueden definir en el propio fichero que las utiliza o en un fichero independiente. Y esta última opción, permite reutilizarlas más fácilmente en distintas aplicaciones. Se dice en este caso, que se está definiendo un *módulo*.

Para construir un módulo se deben realizar dos pasos: primero *definir el módulo*, es decir, describir qué acciones ofrece el módulo y, segundo, *diseñar el módulo*, es decir, describir cómo se llevan a cabo las diferentes acciones.

Para *definir un módulo* en C se crea un fichero de cabeceras (la extensión es, por convenio, `.h`) que contiene la definición de constantes, definición de tipos y los prototipos de las funciones y los procedimientos que lo forman.

Además, hay que *diseñar el módulo*; en otro fichero (normalmente con el mismo nombre que el de cabeceras, pero con extensión `.c`) se escribe el cuerpo de las funciones y procedimientos del módulo. Al principio de este fichero se incluye el fichero de cabecera.

Cuando desde un programa –u otro módulo– se quiera utilizar cualquiera de las funciones, procedimientos, tipos o constantes de dicho módulo, sólo hay que incluir su fichero de cabeceras (`.h`) y compilar el fichero del programa junto con el fichero que contiene el diseño (el de extensión `.c`).

De acuerdo a esto, en el ejemplo se distinguen las siguientes acciones, en función de si sólo servirían para el problema que se pretende resolver, o si se pueden reutilizar:

Acciones genéricas	Acciones propias del problema
	muestraMenu
leerOpcion	opciones
leeRealPositivo	areaCirculo
leePunto	areaCuadrado
calculaAreaCirculo	areaRectangulo
calculaAreaRectangulo	longitudSegmento
calculaLongSegmento	ejecutaOpcion
	muestraResultado

Por ejemplo, `calculaAreaCirculo` es más general que `areaCirculo`: la primera se puede usar siempre que se conozca el radio, mientras que la segunda sólo sirve si se quiere leer el radio desde la entrada estándar. Ese razonamiento se puede aplicar, en menor o mayor grado, a todas las acciones que se han distinguido como *genéricas*. Lo que puede conducir a plantearse el diseño de un módulo para las acciones genéricas (las reutilizables), como el siguiente:

geometria.h

```
1  /* Módulo geometría: Fichero de definición geometría.h */
2  #define PI 3.141592
3
4  float leeRealPositivo();
5  /* Lee de teclado un valor real positivo */
6  /* Pre: no hay datos */
7  /* Post: devuelve num, valor real positivo */
8
9  void leePunto(float *x, float *y);
10 /* Lee de teclados las dos coordenadas de un punto */
11 /* Pre: no hay datos */
12 /* Post: x e y son valores reales */
13
14 float calculaAreaCirculo(float radio);
15 /* Dado el radio, calcula el área de un círculo */
16 /* Pre: radio es un valor real positivo */
17 /* Post: devuelve area,  que contiene un valor real */
18 /* que coincide con el área del círculo */
19
20 float calculaAreaRectangulo(float lado1, float lado2);
21 /* Dados los lados, calcula el área de un rectangulo */
22 /* Pre: lado1 y lado2 son valores reales positivos */
23 /* Post: devuelve area, que es un valor real que coincide */
24 /* con el área de la figura */
25
26 float calculaLongSegmento(float x1, float y1,float x2, float y2);
27 /* Calula la longitud del segmento entre (x1,y1) y (x2,y2) */
28 /* Pre: x1, x2, y1, y2, son reales */
29 /* Post: devuelve long, valor real positivo que es la */
30 /* longitud del segmento */
31
32 int leerOpcion(int min, int max);
33 /*Lee un valor entero que se corresponde con una opción entre min y max*/
34 /*Pre: min y man son enteros,acotan el rango de opciones válidas*/
35 /* Post: devuelve opc, entero entre min y max */
```

Este fichero de cabeceras, se completa con el siguiente fichero de diseño:

geometria.c

```
/* Módulo geometría: Fichero de diseño geometría.c */

#include <math.h>
#include "geometria.h"

float leeRealPositivo() {
  float num;

  scanf("%f", &num);
  while (num<0) {
    scanf("%f", &num);
  }
  return num;
}

void leePunto(float *x, float *y) {
  float cx, cy;

  printf("\n Coordenada x: ");
  scanf("%f",&cx);
  printf("\n Coordenada y: ");
  scanf("%f",&cy);
  *x=cx;
  *y=cy;
}

float calculaAreaCirculo(float radio) {
  float area;

  area=PI*radio*radio;
  return area;
}

float calculaAreaRectangulo(float lado1, float lado2) {
  float area;

  area=lado1*lado2;
  return area;
}

float calculaLongSegmento(float x1, float y1,float x2, float y2){
  float long;
  float a1,a2;

  a1=x1-x2;
  a2=y1-y2;

  long=sqrt(a1*a1+a2*a2);

  return long;
}
```

```
52
53 int leerOpcion(int min, int max) {
54   int opc;
55
56   printf("\n > Opción elegida (entre %d y %d)? ", min, max);
57   scanf("%d", &opc);
58   while (opc<min || opc>max){
59     printf("\n Opción no válida.");
60     printf("\n > Opción elegida (entre %d y %d)? ", min, max);
61     scanf("%d", &opc);
62    }
63   return opc;
64 }
```

Nótese que en la línea 4 del fichero de diseño, `geometria.c`, se añade la directiva `#include "geometria.h"`, indicando que se utilizarán las funciones y constantes *definidas* en ese fichero. Y que, además, no se utiliza exactamente la misma sintaxis que se utiliza en la línea 3 para referirse al módulo de la biblioteca estándar: con los módulos propios se utilizan comillas (") en lugar de los delimitadores < y >.

En el módulo `geometria` no se han incluido todas las acciones que necesita la aplicación; faltan las más dependientes de ésta, que se definirán en el propio fichero del programa.

El programa se puede escribir sin más que encadenar las tres acciones principales del análisis realizado, y sigue el esquema mostrado en la subsección 4.1.3: primero, los ficheros de cabeceras (tanto los de la biblioteca estándar, como el de `geometria`). Después la definición de funciones y procedimientos propios y, después de la función `main`, el diseño de funciones y procedimientos propios. Si se sigue este estilo, se consigue que sea más fácil leer el programa principal.

El programa que se propone es el siguiente, `geometriaPlana.c`:

geometriaPlana.c

```
1  #include <stdio.h>
2  #include <math.h>
3  #include "geometria.h"
4
5  #define AREACIRCULO 1
6  #define AREACUADRADO 2
7  #define AREARECTANGULO 3
8  #define LONGITUDSEGMENTO 4
9
10 /*Definición de prototipos de funciones y procedimientos propios*/
11 void muestraMenu();
12 /* Informa al usuario de sus opciones */
13 /* Pre: no hay datos */
14 /* Post: no hay resultados */
15
16 int opciones();
17 /* Pide opción al usuario: muestra el menú y lee la opción */
18 /* Pre: no hay datos */
19 /* Post: devuelve opc, entero entre 1 y 4 */
```

```
20 float areaCirculo();
21 /* Pide el valor del radio y calcula el área del círculo */
22 /* Pre : no hay datos */
23 /* Post: devuelve area, que contiene un valor real, */
24 /* el área del círculo cuyo radio es radio */
25
26 float areaCuadrado();
27 /* Pide el valor del lado y calcula el área del cuadrado */
28 /* Pre : no hay datos */
29 /* Post: devuelve area, que contiene un valor real,  */
30 /* el área del cuadrado cuyo lado es lado */
31
32 float areaRectangulo();
33 /* Pide el valor de los lados y calcula el área del rectángulo */
34 /* Pre : no hay datos */
35 /* Post: devuelve area, real, y su valor es el área del */
36 /* rectángulo de lados lado1 y lado2 */
37
38 float longitudSegmento() ;
39 /* Lee 2 puntos y calcula la longitud del segmento que definen */
40 /* Pre: no hay datos */
41 /* Post: devuelve long, real positivo cuyo valor es */
42 /* la longitud del segmento */
43
44 float ejecutaOpcion(int op) ;
45 /* Realiza un proceso determinado en función del valor de op */
46 /* Pre: op es un valor entero entre 1 y 4 */
47 /* Post: devuelve res, que es un valor real */
48 /* que coincide con el resultado del proceso elegido */
49
50 void muestraResultado(int op, float resultado) ;
51 /* Muestra el resultado del proceso elegido */
52 /*Pre:op es un valor entero entre 1 y 4 y resultado es un real*/
53 /* Post: no hay resultados */
54
55 /**********************/
56 /* Función principal */
57 /**********************/
58
59 int main() {
60   int m;
61   float resultado;
62
63   /* Muestra el menú y lee una opción */
64   m=opciones();
65
66   /* Ejecuta el subprograma correspondiente */
67   resultado=ejecutaOpcion(m);
68
69   /* Muestra el resultado */
70    muestraResultado(m, resultado);
71 }
72
```

```
73 /* Diseño de funciones y procedimientos propios */
74
75 void muestraMenu(){
76   printf("Introduzca opción:\n");
77   printf("\t %d.- Área de un círculo\n", AREACIRCULO);
78   printf("\t %d- Área de un cuadrado\n", AREACUADRADO);
79   printf("\t %d.- Área de un rectángulo\n", AREARECTANGULO);
80   printf("\t %d.- Longitud de un segmento\n", LONGITUDSEGMENTO);
81 }
82
83 int opciones() {
84   int opc;
85
86   mostraMenu();
87   opc=leerOpcion(1,4);
88   return opc;
89 }
90
91 float areaCirculo() {
92   float area,radio;
93
94   printf("\nValor del radio? ");
95   radio=leeRealPositivo();
96   area=calculaAreaCirculo(radio);
97
98   return area;
99 }
100
101 float areaCuadrado() {
102   float area,lado;
103
104   printf("\nValor del lado?  ");
105   lado=leeRealPositivo();
106   area=calculaAreaRectangulo(lado,lado);
107
108   return area;
109 }
110
111 float areaRectangulo() {
112   float area,lado1,lado2;
113
114   printf("\nValor del primer lado?  ");
115   lado1=leeRealPositivo();
116   printf("\nValor del segundo lado?  ");
117   lado2=leeRealPositivo();
118   area=calculaAreaRectangulo(lado1,lado2);
119
120   return area;
121 }
122
123 float longitudSegmento() {
124   float x1,y1,x2,y2;
125   float long;
```

```
126
127   printf("\n Valor del primer punto? ");
128   leePunto(&x1,&y1);
129   printf("\n Valor del segundo  punto? ");
130   leePunto(&x2,&y2);
131   long=calculaLongSegmento(x1,y1,x2,y2);
132   return long;
133 }
134
135 float ejecutaOpcion(int opc) {
136   float res;
137
138   switch (opc)  {
139    case AREACIRCULO:
140      res=areaCirculo();
141      break;
142
143    case AREACUADRADO:
144      res=areaCuadrado();
145      break;
146
147    case AREARECTANGULO:
148      res=areaRectangulo();
149      break;
150
151    case LONGITUDSEGMENTO:
152      res=longitudSegmento();
153      break;
154    }
155   return res;
156 }
157
158 void muestraResultado(int opc, float resultado) {
159   switch (opc) {
160    case AREACIRCULO:
161      printf("\nEl área del círculo es %f.\n", resultado);
162      break;
163
164    case AREACUADRADO:
165      printf("\nEl área del cuadrado es %f.\n", resultado);
166      break;
167
168    case AREARECTANGULO:
169      printf("\nEl área del rectángulo es %f.\n", resultado);
170      break;
171
172    case LONGITUDSEGMENTO:
173      printf("\nLa longitud del segmento es %f.\n", resultado);
174      break;
175    }
176 }
```

4.3.2. Definición y Uso de Bibliotecas en C.

A modo de resumen, se citan a continuación los pasos en la definición y uso de módulos, en el entorno del compilador `gcc`:

1. Crear el fichero de cabecera, por ejemplo `modulo.h`.
2. Crear el fichero con funciones y procedimientos, `modulo.c`. Una de las primera instrucciones de éste será `#include "modulo.h"`.
3. Compilar el módulo:

   ```
   gcc -c modulo.c
   ```

   Esto produce un fichero con el código objeto del módulo: `modulo.o`.

   Este paso es aconsejable para conseguir un uso más eficiente del módulo (ahorra futuras "recompilaciones").

Para utilizar el módulo (sus tipos, funciones, ...) desde otro programa, se han de seguir los siguientes pasos:

1. Crear el fichero, `mifichero.c`, incluyendo el fichero de cabecera del módulo, `#include "modulo.h"`.
2. Compilar el programa y el módulo:

   ```
   gcc mifichero.c modulo.o
   ```

   Esto genera un fichero con el código objeto del programa `mifichero.o` y lo enlaza junto con el código objeto del modulo para crear un ejecutable, `a.out`.

El siguiente paso, es la creación de una *biblioteca de programas*, que es útil cuando se dispone de varios módulos relacionados (por el tipo de funciones que realizan, por ejemplo) y cuyo uso es bastante común.

Siguiendo con el ejemplo anterior, este primer módulo diseñado, `geometria`, en el futuro, si se sigue enriqueciendo con más funciones y procedimientos (y tipos de datos), podría dar lugar a más de un módulo. Una posible separación sería distinguir entre operaciones puras de geometría (`calculaAreaCirculo`, `calculaAreaRectangulo`...) y operaciones de lectura (`leeRealPositivo`, `leePunto`,...). Podría llegar un momento en que ya se dispusiera de varios módulos de operaciones (y tipos de datos) propias y muy relacionadas (módulos de geometría plana, de geometría espacial, ...) y que fuera conveniente la creación de una biblioteca.

Así, una biblioteca es un fichero que contiene código objeto (proveniente normalmente de varios módulos) y que ofrece la ventaja de que dicho código objeto puede ser enlazado con el código objeto de otros programas que usa una función de esa biblioteca.

Una biblioteca se estructura internamente como un conjunto de módulos objeto (cuya extensión es `.o`). Cada uno de estos módulos puede ser el resultado de la compilación de un fichero de código fuente (extensión `.c`) que puede contener, a su vez, una o varias funciones. La extensión por defecto de los ficheros de biblioteca es `.a` y se acostumbra a que su nombre empiece por el prefijo `lib`.

Si se utiliza el compilador `gcc` para crear una biblioteca se seguirán los siguientes pasos:

1. Crear los ficheros fuentes, con extensión `.c` y los de cabecera `.h` de cada módulo.
2. Compilar dichos ficheros:

   ```
   gcc -c fichero1.c fichero2.c ...
   ```

3. Crear un fichero de cabecera (con la definición de tipos y los prototipos de funciones y procedimientos que ofrece la biblioteca). Este paso es opcional, puesto que cada módulo tendrá su propio fichero de cabecera.
4. Unir los módulos objeto en un sólo fichero. Por ejemplo, en los sistema tipo UNIX se puede utilizar la orden `ar`:

   ```
   ar -r libnombreBiblioteca.a
   ```

Para utilizar una biblioteca (sus tipos, funciones, ...) desde otro programa, se han de seguir los siguientes pasos:

1. Crear el fichero, `mifichero.c`, incluyendo el fichero de cabecera de la biblioteca: `#include "fichLib.h"` o como quiera que se llame.
2. Compilar el programa y enlazarlo con la biblioteca:

   ```
   gcc mifichero.c -lnombreBiblioteca
   ```

   Obsérvese que aunque la biblioteca se llama `libnombreBiblioteca.a`, sólo hay que poner en la opción `-l` la parte `nombreBiblioteca`.

Esto genera un fichero con el código objeto del programa `miFichero.o` y lo enlaza junto con el código objeto que se necesite de la biblioteca para crear un ejecutable, `a.out`.

4.4. Resumen y Consideraciones de Estilo.

El contenido de este tema, está directamente relacionado con la etapa de *planteamiento y planificación de la solución*, la segunda de las que se presentaba en el tema 1 para realizar algoritmos de forma que se cumplan los objetivos de la asignatura; por supuesto, una buena buena planificación también redundará en una mayor simplicidad de la etapa de *formulación*. Asimismo, los comentarios en forma de precondición y postcondición, deberían ayudar a *evaluar la corrección*. Por lo que respecta a la fase de *implementación*, el trabajo a desarrollar en las prácticas, completará el contenido de este tema.

Como recomendaciones de estilo, resaltar las siguientes:

- Al hablar de variables, se hacía notar que elegir nombres significativos puede ayudar a entender mejor un algoritmo. Por supuesto, esto también es aplicable a los identificadores de acciones no primitivas: resulta más fácil entender `leeRealPositivo()` que `P23v2()`.

  Además, se recomienda seguir el mismo convenio que ya se comentó con respecto a cómo nombrar variables: utilizar minúsculas y, en el caso de que el nombre sea compuesto, se utilizar mayúscula al principio de cada nueva palabra.

- En la medida de lo posible es bueno mantener el esquema presentado a lo largo del tema: definición de prototipos *antes* de la función principal y diseño *después* de la función principal. De esta forma, si lo que se pretende es simplemente entender qué hace el programa, la información está agrupada de forma más racional. Si además interesara saber cómo se hace alguna operación concreta, se puede acceder a su código concreto, al final, sin que entorpezca la lectura del programa principal.
- Y, por supuesto, insistir en el uso de comentarios para indicar qué hacen las acciones no primitivas y cuáles son sus precondiciones y postcondiciones. Es de especial relevancia en el fichero de definición y en la parte de definición de prototipos, puesto que así el usuario/lector sabe qué hace la operación y cómo debe usarla. Nótese como se han utilizado los comentarios en el ejemplo: un usuario de `geometria` no tiene por que saber cómo se ha diseñado la operación `calculaAreaCirculo` pero sí que debe saber qué hace y que el dato, `radio`, debe ser un valor real positivo para que la función devuelva un resultado correcto.

4.5. Glosario.

argumento Nombre que reciben también los parámetros de entrada de las funciones.

biblioteca de programas La denominación proviene del inglés, *programs library*, lo que ha conducido a una mala traducción, librería de programas, de uso habitual. Se remarca en este glosario en un intento de combatir esa mala traducción.

cabecera Línea en la que se indica el identificador de una función, su lista de parámetros formales y el tipo del resultado que devuelve.

función Acción no primitiva que, al tener un tipo asociado y un nombre que puede identificarse con el único valor que devuelve, puede utilizarse en expresiones.

instancia Valor concreto que se da a un parámetro formal de entrada.

procedimiento Acción no primitiva.

prototipo Línea de un programa en `C`, en el que se indica la cabecera de una función o procedimiento. Finaliza con el símbolo `;` .

4.6. 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. “El Lenguaje de Programación `C`. Diseño e Implementación de Programas”. Félix García, Jesús Carretero, Javier Fernández & Alejandro Calderón. Pearson Education, Madrid 2002.
4. “The C Programming Language”. Brian W. Kernigham & Dennis M. Ritchie. Prentice-Hall Inc. 1988.

4.7. Problemas Propuestos.

1. Modificar el algoritmo `lecturaCorrecta()` para que sólo admita valores válidos de ángulos de un triángulo: tal y como está definido en el ejemplo, admitiría como valores 180, 0 y 0 (ó 90, 90 y 0) que suman 180.0, pero que no son válidos para definir un triángulo.

2. Modificar el programa `geometriaPlana.c` de forma que el usuario pueda repetir el proceso cuantas veces quiera. Para acabar, dispondrá de una cuarta acción, `finalizar`. Realizar la modificación atendiendo a las acciones no primitivas que se verán afectadas. Volver a especificar todas las acciones no primitivas afectadas.

3. Modificar el programa `geometriaPlana.c` de forma que el usuario pueda calcular un área o bien dando el valor del radio o los lados, o bien indicando los puntos que delimitan radio o lados, entendidos como segmentos. Realizar la modificación atendiendo a las acciones no primitivas que se verán afectadas. Volver a especificar todas las acciones no primitivas afectadas.

4. Enriquecer el módulo `geometria.c` añadiendo las operaciones que permitan calcular el área de un triángulo y la longitud de una circunferencia. Modificar de forma adecuada el fichero `geometria.h` para que las nuevas acciones sean utilizables. Especificar las nuevas acciones.

5. Utilizar el módulo `geometria` de forma que se pueda utilizar en la definición y diseño de un nuevo módulo `geometriaEspacial` que permita calcular el volumen de un cono, el volumen de un cilindro, el volumen de un tetraedro, el volumen de un pirámide de base rectangular y el volumen de un paralelepípedo.

6. Utilizar la acción `maxComDiv()`, para realizar un programa en el que un usuario podrá calcular el máximo común divisor de una serie de números que irá introduciendo por teclado. La serie debe tener al menos dos números y finalizará cuando el número introducido sea 0. Sólo se permitirá leer números positivos (y el 0 final, evidentemente). Especificar el algoritmo.

7. El siguiente algoritmo:

```
int X(int a, int n){
  int y, i;

  y=1;
  for(i=1; i<=n; i++){
     y=y+a*a;
  }
}
```

   ¿calcula $\sum_{i=1}^{n} a^2$? Razona la respuesta.

8. Escribe todo lo que se podría leer en la pantalla del ordenador al ejecutar el siguiente fragmento de programa, sabiendo que al ejecutarlo se dará a `num` el valor `182745`:

```
#include <stdio.h>

int funcionAux(int numero);
int main() {
  int num, res, aux;

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

```
  res = 0;
  aux = 1;
  while ((num/aux)>0) {
    printf("Printf (1): %d\n",funcionAux(num/aux));
    printf("Printf (2): %d\n",num);
    aux = aux * 10;
    res = res + funcionAux(num/aux);
  }
  printf("Printf (3): %d\t %d\n", num, res);
}
int funcionAux(int numero) {
   int res;
   res = 0;
   while (numero>0) {
      if (numero %2==0)
         res=res+numero %10;
      numero=numero/10;
   }
  return res;
}
```

9. Escribir una función o procedimiento (**justificar la elección**) que, dado un valor entero indique si ese valor es o no es una **potencia completa**.

   Se dice que un entero `n` es una *potencia completa* si se cumple que existe un entero `p` que es un **divisor primo** de `n`, y, además, p^2 también es divisor de `n`.

   Por ejemplo, el `81` sería una potencia completa, ya que `3` es divisor de 81, `3` es primo y, además, 3^2 es `9`, que también es divisor de `81`.

   Se debe indicar cuál es la precondición y cuál la postcondición. Además, para considerar correcta la respuesta, el código debe escribirse utilizando de forma adecuada la función, que YA ESTÁ DEFINIDA, `esPrimo`, que responde al prototipo,

   ```
   boole esPrimo(int n);
   /* Pre: n, entero positivo */
   /* Post: devuelve cierto si n es primo,
         falso si no lo es */
   ```

10. Sabiendo que YA ESTÁN DEFINIDAS las funciones `esPrimo` y `elevar`, que responden a los prototipos,

    ```
    boole esPrimo(int n);
    /* Pre: n, entero positivo */
    /* Post: devuelve cierto si n es primo, falso si no lo es */

    int elevar(int k);
    /* Pre: k, entero positivo */
    /* Post: devuelve 2^{k} */
    ```

    y suponiendo YA DEFINIDO el tipo `boole`, se pide lo siguiente:

    a) Escribir la función `numDivPrimos`, que debe responder al prototipo,

    ```
    int numDivPrimos(int m);
    /* Pre: m, entero positivo */
    /* Post: devuelve el numero de divisores primos de m */
    ```

En la definición de esta función DEBE utilizarse correctamente la función `esPrimo`.

b) La α-imagen de n es $\alpha(n) = 2^{T-1}$, siendo T el número de divisores primos de n.
Escribir un programa que lea una secuencia de valores enteros por teclado y, por cada valor leído, calcule su α-imagen y la muestre por pantalla. Cuando se lea el valor `0` finaliza la introducción de datos.
Se DEBE razonar cuáles de entre las funciones definidas en el enunciado y en el apartado anterior son útiles en el desarrollo del programa y USARLAS convenientemente en el diseño del programa.

11. El método de la bisección (ver figura 4.1) permite encontrar un cero de la función $y = f(x)$, continua en el intervalo $[a, b]$ y tal que $f(a)$ y $f(b)$ tienen distinto signo ($(f(a)f(b) < 0)$.

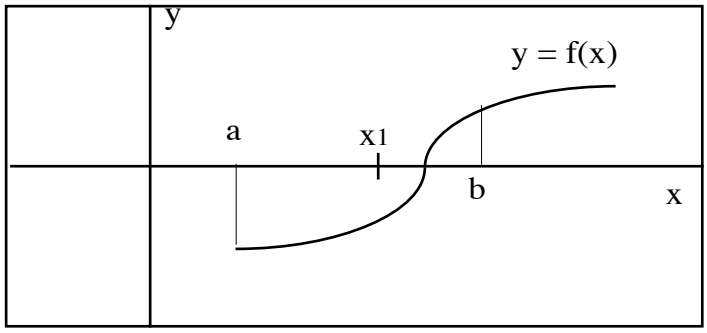


Figura 4.1: Método de la bisección

El método consiste en dividir el intervalo en dos partes iguales. Si llamamos x1 a la mitad del intervalo, se comprueba el signo de $f(x1)$: si $f(x1)$ tiene el mismo signo que $f(a)$ se continúa la búsqueda en el intervalo $[x1, b]$ y si $f(x1)$ tiene el mismo signo que $f(b)$, la búsqueda se continúa en el intervalo $[a, x1]$. Se repite el proceso para el intervalo resultante, hasta que se llega a que f vale 0 ó que la longitud del intervalo es menor que un cierto valor, epsilon.

Suponiendo que existe una acción no primitiva ya definido,

`float` calculaFuncion (`float` x);

que calcula el valor de la función f en el punto x, escribir un algoritmo que resuelva el cero de una función $f(x)$.

12. Un número entero N se dice que es **curioso** o **automórfico**, cuando N^2 acaba en N. Por ejemplo:

- el 5, $5 \times 5 = 2\mathbf{5}$,
- el 6, $6 \times 6 = 3\mathbf{6}$,
- el 25, $25 \times 25 = 6\mathbf{25}$,
- el 376, $376 \times 376 = 141\mathbf{376}$,
- el 109376, $109376 \times 109376 = 11963\mathbf{109376}$,...

a) Escribir una función que determine si un número es o no es curioso. La cabecera debe ser:

```
boole esCurioso (int n)
```

Hay que indicar la precondición y la postcondición, además del cuerpo de la función.

b) Escribir un programa que use la función del apartado anterior para determinar cuántos números curiosos pares y cuántos números curiosos impares hay entre 1 y un valor entero K, que se leerá de teclado.

13. Diremos que dos números enteros positivos `n` y `m` están **liados** si al pasarlos a binario la cantidad de `1`'s de `n` es igual a la cantidad de `0`'s significativos de `m` y la cantidad de `0`'s significativos de `n` es igual a la cantidad de `1`'s de `m`.

    Por ejemplo, el número 50 (110010) está liado con: el 35 (100011), el 37 (100101), el 38 (100110), el 41 (101001), el 42 (101010), el 44 (101100), el 49 (110001), el 52 (110100) y el 56 (111000).

    a) Escribir una función que determine si dos números están o no liados, indicando su precondición y su postcondición. La cabecera debe ser:

    ```
    boole liados (int n, int m)
    ```

    b) Escribir un programa que use la función del apartado anterior para determinar cuáles son los números liados con uno dado, que se leerá de teclado.

    Nota: Dado `n` ¿cuál es el rango de los posibles candidatos a estar liados con él?

14. ¿Cuál es el dominio de definición del siguiente algoritmo? Justifica la respuesta.

```
char mayuscula(char c) {

  return c-32;
/*Post:devuelve c en mayúsculas*/
}
```

    NOTA: Para transformar 'z' (código ASCII 122) en 'Z' (código ASCII 90), por ejemplo, basta con restarle 32. Los códigos ASCII de las letras mayúsculas van de 65 a 90.

15. El siguiente algoritmo determina si en un número aparece o no un determinado dígito ¿Cuál es su dominio de definición? Justificar la respuesta.

```
boole contieneDigito (int num, int digito) {
  int n;

  n = num;
  while ((n>9) && (n%10!=digito)) {
    n=n/10;
  }

  return (n%10==digito);
}
```

16. Simplificar al máximo la siguiente función y especificarla:

```
int simplFuncion(int y, int z, int v) {
   int i, w, x;

   x = 0;
   if (x > 1) {
      x = 2;
```

```
   }
   else {
      x = 1;
   }
   if (x == 1) {
      x = y;
      x = (x+z)/2;
      w = x ;
      for (i=1;i<=((w)/1)+1);i++) {
          v = v*x+v;
      }
      v = v/w;
   }
   else {
      x = z;
      x = (x+z)/2;
      w =  x;
   }

   return w;
}
```

17. Simplificar al máximo este procedimiento, comentando brevemente todas las simplificaciones realizadas (Nota: N > i > 0).

```
void simplProcedimiento(int N, float b, int *i, float *a) {
    int j,k;
    float w,v;

    j = *i;
    while (j <= N) {
      v = j;
      *a = b + (j-i);
      w = sqrt(((*a)*(*a) + b*b)/2);
      while ((*a < b) || (j == *i)) {
        v = w - b;
        *a = w;
        *i = *i + 1;
      }
      if (v >= 0) {
         for (k=i; k<=j+1; k++) {
             a = a + k;
             *i = j;
         }
      }
      else {
         for (k=i; k<=j+1; k++) {
             v = w + b;
         }
      }
      j = j + 1;
   } /* fin del while (j <= N) */
}
```

18. Dado el siguiente algoritmo, que calcula el resto de la división entera, ¿cuál es su dominio de definición?

```
int resto (int numer, int denom) {
/* Pre: ..... */

  while (numer >= denom)
    numer = numer - denom;
  return numer;
/* Post: devuelve el resto de la división
entera entre numer y denom */
}
```

19. Se pretende escribir una función que resuelva la conjetura de los capicúas: dado un entero positivo, hay que indicar cuántas veces se debe repetir el proceso de sumarlo al valor obtenido al invertir sus dígitos, antes de obtener un resultado capicúa.

    Indicar cuáles son los 4 errores (**no sintácticos**) cometidos al desarrollar la siguiente función, justificando para cada error el porqué:

```
int conjetura (int num) {
/* Pre: num contiene un valor positivo */
   int veces,aux,suma=0;

   veces=0;
   aux=num;
   do {
   /* ... hasta llegar a un capicua */
      while ((aux/10) > 9) {
         suma=(suma*10)+(aux%10);
         aux=aux/10;
      }
      suma=(suma*10)+aux;
      /* suma vale num invertido */
      /* si el número no es capicúa, repito */
      if (num!=suma) {
         veces=veces+1;
         num=num+suma;
      }
   } while (num!=suma);

/* Post: devuelve el número de veces que se */
/* repite el proceso */
}
```