2.5a. Para llevar a cabo este programa se propone el siguiente esquema:

```
cout << "Estate atento" << endl;
⟨Esperar un instante aleatorio⟩
while (⟨no se haya pulsado ninguna tecla⟩) {
  cout << '*' << flush;
  ⟨Esperar un lapso de tiempo⟩
}
```

Si no sabes a dónde recurrir para calcular un número aleatorio, o para detener la ejecución del programa durante un cierto tiempo, consulta la pista 2.5b.

2.5b. Para detallar este ejercicio necesitamos recurrir a la biblioteca de subprogramas de nuestro lenguaje en busca de dos métodos: uno para detener la ejecución un periodo de tiempo y otro para consultar si se ha pulsado una tecla. La generación de números aleatorios está explicada en el ejercicio 3.13.

Espera El procedimiento

```
void usleep(unsigned long microsegundos);
```

que está en el fichero de cabecera `unistd.h`, detiene la ejecución del programa durante los microsegundos especificados.

Pulsación En Unix (el contexto en el que casi siempre vive C), el teclado es un *dispositivo* de entrada. Hay varias funciones que nos permiten saber en qué dispositivos hay datos listos para leer. Si tenemos datos en el dispositivo del teclado será porque se ha pulsado una tecla. La función más simple para este propósito es ésta,

```
int poll(pollfd ds[], unsigned int nds, int espera);
```

que aparece en el fichero de cabecera `sys/poll.h`. El array `ds` contiene los dispositivos que deseamos vigilar; `nds` es la longitud del array anterior; y `espera` son los milisegundos que queremos esperar a que ocurra algo en los dispositivos de `ds`. Si `espera` es negativo, se espera indefinidamente; si es 0, se mira el estado actual de los dispositivos y termina. El resultado de esta función es el número de dispositivos en los que ocurrió algo. El tipo `pollfd` es un registro con dos campos interesantes: `fd` es el número del dispositivo que queremos consultar (0 para el teclado) y `events` que, para indicar que estamos interesados en leer, debe valer `POLLIN`. El siguiente código espera a que pulsemos una tecla:

```
pollfd pulsacion[1];
pulsacion[0].fd = 0;
pulsacion[0].events = POLLIN;
poll(pulsacion, 1, -1);
```

2.6a. Tanto el cifrado como el descifrado tienen que basarse en una *clave*, que será capaz de describir el código de rotación que estamos utilizando. Una forma sencilla de representar dicha clave es indicando cuál es el cifrado de la primera letra del alfabeto. Por ejemplo, en el caso del código de Julio César, la clave sería la letra D que es la letra en la que ciframos la A; con este dato podemos conocer el cifrado de cualquier otra letra. Otra forma sencilla para representar la clave es con un número entero que indique cuánto hay que avanzar (o retroceder) por el alfabeto para cifrar una letra. En este caso, la clave para

el código de Julio César sería 3, ya que cualquier letra es sustituida por la que está tres posiciones más allá en el alfabeto.

2.6b. Es muy importante para la corrección del programa fijar convenientemente el alfabeto sobre el que actúa el código. Al menos en un primer intento, no te compliques y elige un alfabeto sencillo.

2.8a. Tenemos que evaluar los datos que recibimos en una sola pasada, ya que sólo podemos leer una vez la entrada estándar. Por tanto, en principio no podemos usar la potencia x^n, puesto que no conocemos aún dicha n. La regla de Horner, debida al matemático inglés Willian George Horner (1786–1837), permite calcular un polinomio de grado n de forma acumulativa:

$$((\ldots((a_0x + a_1)x + a_2)x + \ldots)x + a_{n-1})x + a_n.$$

2.13a. Los diversos parámetros que pueden modificar el patrón de escritura son: el espacio en blanco inicial para la primera línea; la anchura total de las líneas; el número de líneas en blanco hasta que aparece la primera línea a la izquierda; y el número de caracteres en los que decrece o aumenta cada línea con respecto a la anterior.

2.17a. La fórmula de la varianza parece indicar que necesitamos hacer dos pasadas sobre los datos: una para calcular primero la media y otra para aplicar esa fórmula. Porque no es razonable pedir a quien use nuestro programa que escriba dos veces los mismo datos, parece que necesitamos almacenarlos internamente. Afortunadamente, un poco de esfuerzo mental y un poco de aritmética básica te permitirán reescribir la fórmula de la varianza, de forma que se pueda calcular de una sola pasada y puedas evitar todas las complejidades que produce el almacenamiento de unos datos arbitrariamente largos.

2.18a. Para poder realizar los cálculos que se indican en el enunciado del ejercicio necesitamos descomponer el número de 9 cifras que se recibe como dato en cada uno de los dígitos que lo componen. En el ejercicio 2.21 puedes encontrar más información.

2.19a. No te obsesiones con la estética. Tres guiones (`---`) es una buena aproximación al *yang*; al quitar el intermedio (`- -`) obtenemos el *yin*.

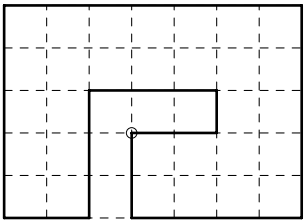
2.19b. La dicotomía entre el *yin* y el *yang* se resuelve con un *bit*. Una vez elegida una correspondencia entre estos principios humanos y los principios informáticos 0 y 1, un hexagrama se reduce a la representación binaria de un número de 6 bits, es decir, un número entre 0 y $2^6 - 1$ (= 63).

2.21a. En los apartados anteriores se pide que el programa lea cantidades que utilizan símbolos que no pueden escribirse con los teclados habituales. Adopta la conversión de símbolos que consideres más adecuada. Por ejemplo podemos proponerte la siguiente:

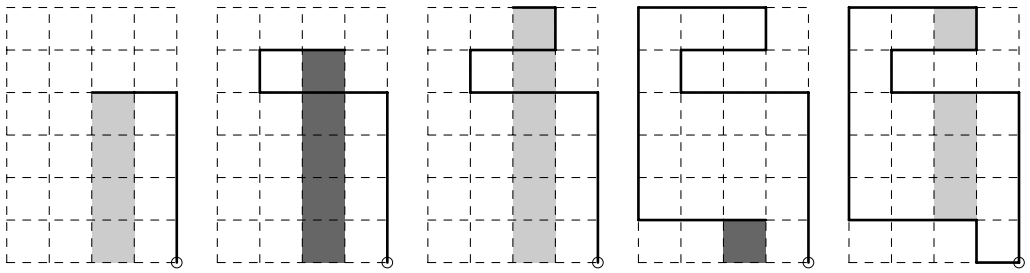
símbolos aztecas				símbolos arameos					
•									
.	p	f	@	1	n	h	?	N	*

Con respecto a los sistemas posicionales, si se considera una base $n \leq 10$, lo habitual es utilizar los dígitos entre 0 y $n - 1$ para expresar los números en dicha base. Si utilizamos una base $n > 10$, suelen añadirse las letras necesarias del alfabeto tomadas en orden. Por ejemplo, para base 13, se consideran números que contiene los trece *dígitos* siguientes: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C.

2.26a. No se debe suponer que el polígono queda por encima del punto de inicio, pero esta suposición puede facilitar la búsqueda de una primera solución. Ten cuidado de no trivializar la solución; se pueden producir polígonos arbitrariamente complejos; basta con tener la paciencia de enumerar `BBDDDDSSSSSIIIIIIIBBBBBDDSSSDDDBII` para conseguir éste:



2.26b. Fíjate en la siguiente figura:



2.27a. No te dejes llevar por la expresión que calculaste para el ejercicio 1.19. Hay una relación muy simple entre un número y el que está a su derecha o inmediatamente debajo. Descúbrela y escribe un programa que calcule el valor de una casilla a partir del valor de la casilla anterior.

2.29a. En ninguno de los casos es necesario suponer conocida la función inversa, es decir, la exponenciación en base b, sino que basta con usar la operación raíz cuadrada.

2.30a. Véase la pista 2.32a.

2.31a. Se trata de simplificar al máximo una fracción $\frac{n}{d}$. Para ello, tenemos que calcular el máximo común divisor del numerador y del denominador, $\mathrm{mcd}(n, d)$, que es el mayor número por el que podemos dividir tanto n como d.

2.32a. La parte entera de un real r es el único entero n que verifica lo siguiente:

$$n \leq r < n + 1.$$

Aplicándolo a este caso tendremos esta equivalencia:

$$\mathit{raíz} \leq \sqrt{r} < \mathit{raíz} + 1 \qquad \Leftrightarrow \qquad \mathit{raíz}^2 \leq r < (\mathit{raíz} + 1)^2.$$

21 Signo del producto

55

Encontrar una solución suficientemente buena o elegante es cosa de paciencia. Lo interesante es buscar *la mejor*, por supuesto desde una perspectiva estética. Daremos el signo como uno de los enteros -1, 0 o 1, según el producto sea negativo, cero o positivo, respectivamente. Ésta es nuestra mejor solución:

```
int signo;
if ((n == 0) || (m == 0)) signo = 0;
else if ((n < 0) == (m < 0)) signo = 1;
else signo = -1;
```

La primera condición captura cuándo el resultado es 0: precisamente si uno de los dos factores es 0. La segunda condición recoge el concepto de *tener el mismo signo*. La expresión `n < 0` pregunta si `n` es negativo. Su resultado es `true` si `n` es negativo y `false` si es positivo. Por tanto, los resultados de dos expresiones de esta forma, una sobre la variable `n` y otra sobre `m`, son iguales precisamente si el signo de ambas variables es el mismo.

22 Ponle tú el título

55

El siguiente bucle,

```
do {
  cin >> n >> m;
} while ((n < 0) || (m < 0));
```

realiza la lectura de dos valores numéricos hasta que ambos sean enteros no negativos. A continuación, el siguiente bucle,

```
while (m != 0) {
  if (m % 2 != 0) x = x + n;
  n = n + n;
  m = m / 2;
}
```

se ejecuta mientras el contenido de `m` sea distinto de cero. En este caso, si `m` es impar acumula en la variable `x` el valor de `n`, tras lo que, independientemente de que `m` sea o no impar, duplica el valor de `n` y reduce a la mitad entera el valor de `m`. Cuando se produce la salida del bucle, en la variable `x` queda almacenado el producto de `m` y `n`. Veamos su funcionamiento con un ejemplo. Supongamos que, inicialmente, el valor de `m` es 5 y el de `n` 3. Los valores que irán tomando `m`, `n` y `x` tras cada ejecución del cuerpo del bucle `while` quedan reflejados en la siguiente tabla:

m	n	x
5	3	0
2	6	3
1	12	**15**
0	24	

Existen diferentes algoritmos para multiplicar dos números enteros sin utilizar el operador producto "`*`"; el que implementa este ejercicio es uno de ellos y recibe el nombre de *multiplicación a la rusa*. En `http://mathforum.org/dr.math/faq/faq.peasant.html` puedes encontrar más información.

2 4 Lectura de Rayuela 56

La solución a este ejercicio es tan trivial como parece: basta con recorrer el archivo del texto original e ir distribuyendo las líneas, de una en una,

una impar y otra par;
una impar y otra par;
...

y así mientras haya líneas en el archivo original; esto es:

```
while (⟨hay líneas en el archivo fuente⟩) {
  ⟨Leer una línea y pasarla al archivo de las "impares"⟩
  ⟨Leer una línea y pasarla al archivo de las "pares"⟩
}
```

El único detalle que empaña tal simplicidad es la posibilidad de que el número de líneas sea impar, con lo que cada línea par que se vaya a leer debe estar precedida por la correspondiente comprobación:

```
while (⟨hay líneas en el archivo fuente⟩) {
  ⟨Leer una línea "impar", y pasarla al archivo de las "impares"⟩
  if (⟨hay líneas en el archivo fuente⟩) {
    ⟨Leer una línea "par", y pasarla al archivo de las "pares"⟩
  }
}
```

El bucle anterior se concreta en nuestro lenguaje de programación como sigue:

```
#include <string>
#include <fstream>
int main() {
  ifstream fuente("rayuela.txt");
  ofstream pares("impares.txt");
  ofstream impares("pares.txt");
  while (!fuente.eof()) {
    string linea;
    getline(fuente, linea); pares << linea << endl;
    if (!  fuente.eof()) {
      getline(fuente, linea); impares << linea << endl;
    }
  }
}
```

El archivo `rayuela.txt` se supone creado y situado en el directorio de trabajo.

2 10 Representación gráfica de funciones 59

Como podemos ver se ha trazado una cabecera, con los límites de la representación de las ordenadas (en la figura `-0.90` y `0.90`), el nombre de la función representada (`y = sen (x)` en el ejemplo) y una línea horizontal de separación. Debajo, para cada línea, se ha escrito el valor de la abscisa (`0.50`, `0.90`,...,`6.50`) correspondiente, una línea vertical para representar un fragmento de eje, y un asterisco para representar la posición de la función. Si la función se sale fuera de la zona de representación, se ha escrito un símbolo "<" o ">", según caiga por la izquierda o por la derecha. Así pues, el programa consta de cuatro pasos consecutivos:

⟨*Pedir los datos* $x_{mín}$, $x_{máx}$, $y_{mín}$, $y_{máx}$⟩
⟨*Trazar la cabecera de la gráfica*⟩
⟨*Trazar las líneas sucesivas*⟩
⟨*Trazar el pie de la gráfica*⟩

La lectura de los datos es trivial, así como el trazado del pie de la gráfica. La cabecera debe reflejar el intervalo de ordenadas elegido, y escribir un eje del tamaño $númY$, centrando la función. Sólo el trazado de cada línea requiere alguna atención, y se puede desglosar así:

⟨*Hallar la abcisa* x_i⟩
⟨*Hallar la posición (en la pantalla) de la ordenada* $f(x_i)$⟩
⟨*Escribir la línea, comprobando si cae fuera de la zona*⟩

Los ajustes entre las coordenadas reales y las de la pantalla se detallan a continuación:

- La abcisa x_i se halla fácilmente:

$$x_i = x_{mín} + i\frac{x_{máx} - x_{mín}}{númX}, \text{ para } i \in \{0, \ldots, númX\}.$$

- Para cada ordenada $y_i = f(x_i)$, su posición será un entero de $posY_i \in \{0, \ldots, númY\}$ cuando $y_i \in [y_{mín}, y_{máx}]$, lo que se consigue fácilmente haciendo que $posY_i$ sea el entero más próximo a

$$númY\frac{y_i - y_{mín}}{y_{máx} - y_{mín}}.$$

- Por último, un valor de $posY_i$ negativo o nulo indica que la función se sale por la izquierda del fragmento del plano representado, mientras que un valor mayor que $númY$ significa que se sale por la derecha, lo que se indica en la pantalla mediante el símbolo "<" o ">" al principio o al final de la línea, respectivamente.

Juntando todo esto, tenemos el siguiente programa completo:

```
#include <iostream>
#include <iomanip>
#include <math.h>
int main() {
  int const numeroX = 16; // tamaño (abcisas) de la rejilla de dibujo
  int const numeroY = 70; // tamaño (ordenadas) de la rejilla de dibujo
  // Lectura de los datos (tamaño de la zona de interés):
  float xMin, xMax,   // abcisas que limitan la región que se va a dibujar
        yMin, yMax;   // ordenadas que limitan la región que se va a dibujar
  cout << "xMín, xMáx:  "; cin  >> xMin >> xMax;
  cout << "yMín, yMáx:  "; cin  >> yMin >> yMax;
  // Cabecera: el eje superior (de ordenadas) de referencia:
  cout << setw(9) << setprecision(2)  << yMin
       << "              y = sen(x)          ";
  cout << setw(32) << setprecision(2) << yMax << endl;
  cout << "------+------------------------------"
       << "---------------------------------------->" << endl;
  // Las líneas de la gráfica, con cada uno de los puntos (abcisa, ordenada):
  float const delta = (xMax-xMin)/numeroX; // distancia (fija) entre abcisas
```

```
  for (int i = 0; i <= numeroX; i++) {
    float xi = xMin + i * delta; // valor de la abcisa que se va a pintar
    cout << setw(5) << setprecision(2)
         << xi << " | ";
    int yi = (int)rint(((sin(xi) - yMin) / (yMax - yMin)) * numeroY);
    if (yi < 1) {
      cout << "<" << endl;
    } else if (yi > numeroY) {
      cout << setw(numeroY) << ">" << endl;
    } else {
      cout << setw(yi)   << "*" << endl;
    }
  }
  // El pie de la gráfica:
  cout << " |" << endl
       << "  x   V" << endl;
}
```

2.15 Los cubos de Nicómaco

62

Utilizaremos la variable `impar` para que vaya tomando los valores de los números impares. Su valor inicial será -1 ya que así, al incrementarla sucesivamente en 2 unidades, se irán generando los valores impares. El valor de n lo pediremos por teclado almacenándolo en la variable `n`:

```
int impar = -1;
int n;
cin >> n;
```

Debemos calcular los `n` primeros cubos:

```
for (int i = 1; i <= n; i++) {
  ⟨Cálculo del cubo i-ésimo a partir de i números impares⟩
}
```

Sabemos que el primer cubo se calcula sumando el primer impar; el cálculo del cubo `i`-ésimo, cuando `i` > 1, se realiza sumando los `i` siguientes impares; necesitaremos, por lo tanto, además de ir generando números impares consecutivos (`impar = impar + 2`), una variable que vaya acumulando su suma (`suma = suma + impar`) y un bucle que controle que estas operaciones se realizan el número adecuado de veces. Añadiendo las instrucciones de salida para que el resultado tenga un formato similar al del enunciado tenemos:

```
impar = impar + 2;
int suma = impar;
cout << i << "^3 = " << impar;
for (int j = 2; j <= i; j++) {
  impar = impar + 2;
  cout << " + " << impar;
  suma = suma + impar;
}
cout << " = " << suma << endl ;
```

216 Un bonito triángulo

62

El triángulo consta de 10 líneas y cada línea está formada por lo siguiente:

- Un número variable de espacios que comenzando en valor 9 va decreciendo una unidad en cada línea hasta llegar a la última con valor 0.
- Una secuencia de dígitos en orden consecutivo creciente, a excepción de la primera línea formada por un único dígito.
- Una secuencia de dígitos en orden consecutivo decreciente, salvo la primera línea.

La construcción del triángulo se puede plantear así:

```
for (int i = 1; i <= 10; i++) {
  ⟨Escribir la secuencia de caracteres espacio⟩
  ⟨Escribir la secuencia creciente⟩
  ⟨Escribir la secuencia decreciente⟩
  cout << endl;
}
```

El refinamiento de la solución anterior puede conducirnos al siguiente programa:

```
char const espacio = ' ';
for (int i = 1; i <= 10; i++) {
  // Escribir la secuencia de caracteres espacio:
  for (int j = 1; j <= 10 - i; j++) {
    cout << espacio;
  }
  // Escribir la secuencia creciente:
  int n = i;
  for (int j = 1; j <= i; j++) {
    if (n == 10) n = 0;
    cout << n;
    n++;
  }
  // Escribir la secuencia decreciente:
  n = n - 2;
  for (int j = 1; j <= i-1; j++) {
    if (n == -1) n = 9;
    cout << n;
    n--;
  }
  cout << endl;
}
```

217 Varianza

63 78

La ecuación usual de la varianza es particularmente desconsiderada en lo que respecta a su implementación. Con esa ecuación difícilmente se calcula la varianza en una sola pasada, que sería lo deseable. Pero si se desarrolla queda:

$$\frac{1}{n-1}\sum_{i=1}^{n} x_i^2 - \frac{1}{n(n-1)}\Big(\sum_{i=1}^{n} x_i\Big)^2.$$

Esta fórmula se implementa fácilmente en una sola pasada:

```
#include <iostream.h>
int main() {
  double accMedia = 0, accMedia2 = 0;
  double x;
  int n = 0;
  cin >> x;
  while (x != 0) {
    accMedia = accMedia + x;
    accMedia2 = accMedia2 + x*x;
    n++;
    cin >> x;
  }
  double varianza = accMedia2 / (n-1) - (accMedia*accMedia) / (n*(n-1));
  cout << "\nVarianza = " << varianza << endl;
}
```

2.20 Suma marciana

64

Una posibilidad consiste en producir cada una de las combinaciones posibles,

♣	♢	♠	♡
0	0	0	0
0	0	0	1
...	...	...	...
0	0	0	9
0	0	1	0
...	...	...	...
0	9	9	9
1	0	0	0
...	...	...	...

y examinar cuáles de ellas verifican esa cuenta. Podemos describir ese proceso así:

```
for (⟨cada valor posible del trébol⟩) {
  for (⟨cada valor posible del diamante⟩) {
    for (⟨cada valor posible de la pica⟩) {
      for (⟨cada valor posible del corazón⟩) {
        if (⟨la combinación verifica la cuenta escrita en la roca⟩) {
          ⟨Escribir esa combinación⟩
        }
      }
    }
  }
}
```

Para concretar un poco más el problema, tendremos en cuenta lo siguiente:

- Por estar en el sistema de numeración decimal, los posibles valores de cada uno de los símbolos que intervienen en la cuenta son los valores del cero al nueve.

- Como se ha empleado el sistema de numeración decimal, la grafía ♣♢♠ representa la cantidad $100\clubsuit + 10\diamondsuit + \spadesuit$.

Entonces, el algoritmo descrito se traduce a C++ fácilmente así:

```
for (int trebol = 0; trebol < 10; trebol++) {
  for (int diamante = 0; diamante < 10; diamante++) {
    for (int pica = 0; pica < 10; pica++) {
      for (int corazon = 0; corazon < 10; corazon++) {
        int const sumando1 = 100*trebol + 10*diamante + pica;
        int const sumando2 = 10*trebol + corazon;
        int const suma = 100*diamante + 10*pica + trebol;
        if ((sumando1 + sumando2) == suma) {
          // Generar la solución:
          cout << "   " << trebol << diamante << pica << endl;
          cout << " + " << ' ' << trebol   << corazon << endl;
          cout << " -----" << endl;
          cout << "   " << diamante << pica << trebol << endl;
          cout << endl;
        }
      }
    }
  }
}
```

Ahora, pueden hacerse dos observaciones que nos permiten limitar un poco los tanteos:

- Los símbolos ♣ y ♢ no pueden ser nulos, ya que están al principio de los números.
- Los cuatro símbolos empleados por los habitantes de Marte pueden suponerse distintos.

Intercalando estas restricciones en el programa anterior, se tiene el siguiente fragmento de programa:

```
for (int trebol = 1; trebol < 10; trebol++) {
  for (int diamante = 1; diamante < 10; diamante++) {
    if (trebol != diamante) {
      for (int pica = 0; pica < 10; pica++) {
        if (trebol != pica && diamante != pica) {
          for (int corazon = 0; corazon < 10; corazon++) {
            int const sumando1 = 100*trebol + 10*diamante + pica;
            int const sumando2 = 10*trebol + corazon;
            int const suma = 100*diamante + 10*pica + trebol;
            if (trebol != corazon && diamante  != corazon
                && pica != corazon && (sumando1 + sumando2) == suma) {
              ⟨Generar esta solución⟩
            }
          }
        }
      }
    }
  }
}
```

Volcán Para trazar el dibujo descrito, usaremos un bucle que escribe una línea en cada vuelta,

```
for (int i = 1; i <= numFilas; i++) {
  ⟨Dibujar la fila i⟩
}
```

donde el trazado de la fila i-ésima consiste en

⟨*Poner los blancos de la izquierda*⟩
⟨*Poner los asteriscos*⟩
⟨*Salto de línea*⟩

El fragmento de ⟨*Poner los asteriscos*⟩ es fácil, porque en cada fila hay el doble que en la anterior, de forma que, partiendo de `numAst = 1` antes de la primera fila, basta con duplicar el número de ellos cada vez y ponerlos, uno a uno,

```
int main() {
  int numFilas;
  cout << "Dame el número de filas:  " << flush;
  cin >> numFilas;
  int const centro = 4 + (1 << (numFilas-1));
  int numAst = 1; // mitad de los asteriscos de cada fila
  for (int i = 1; i <= numFilas; i++) { // Dibujar la fila i:
    // Poner los blancos de la izquierda:
    repetirCaracter(' ', centro - numAst);
    // Poner los asteriscos:
    numAst = numAst * 2;
    repetirCaracter('*', numAst);
    // Salto de línea:
    cout << endl;
  }
}
```

donde `repetirCaracter` se puede encapsular como un subprograma trivial:

```
void repetirCaracter(char const c, int const n) {
  for (int j = 1; j <= n; j++) {
    cout << c;
  }
}
```

Mosaico La cosa consiste en escribir ocho filas:

```
for (int i = 1; i <= tamanyo; i++) {
  ⟨Dibujar la fila i⟩
}
```

En cada línea `i`, los caracteres `j` (de 1 a `tamanyo`) son blancos o negros dependiendo de la paridad de `i+j`. Cada línea acaba con un salto a la siguiente:

```
for (int j = 1; j <= tamanyo; j++) {
  if ((i+j) % 2 == 0) cout << " ";
```

```
  else cout << "*";
}
cout << endl;
```

El programa completo queda como sigue:

```
#include <iostream>
int main() {
  int const tamanyo = 8;
  for (int i = 1; i <= tamanyo; i++) {
    // Dibujar la fila i:
    for (int j = 1; j <= tamanyo; j++) {
      if ((i+j) % 2 == 0) cout << " ";
      else cout << "*";
    }
    cout << endl;
  }
}
```

Tablero Considerando el programa anterior como punto de partida, el cambio que se propone tiene dos efectos: uno consiste en repetir cada fila el número de veces que indique el `ancho` de los escaques, de forma que ahora, cada hilera de escaques se compone de `ancho` líneas de asteriscos, logrando así que los escaques tengan la altura deseada; el segundo efecto consiste en que, en cada fila, los blancos y los asteriscos se pintan de `ancho` en `ancho`, en vez de uno a uno, y así se logra el efecto ensanchador. En resumen, el programa completo es el siguiente:

```
#include <iostream>
void repetirCaracter(char const c, int const n) {
  for (int i = 1; i <= n; i++) {
    cout << c;
  }
}
int main() {
  int const tamanyo = 8;
  int ancho;
  cout << "Ancho del escaque:  ";
  cin  >> ancho;
  for (int i = 1; i <= tamanyo; i++) {
    for (int ii = 1; ii <= ancho; ii++) {
      for (int j = 1; j <= tamanyo; j++) {
        if ((i+j) % 2 == 0) {
          repetirCaracter(' ', ancho);
        } else {
          repetirCaracter('*', ancho);
        }
      }
      cout << endl;
    }
  }
}
```

Para resolver este ejercicio supondremos que el polígono está situado por encima del punto de origen y que su interior queda a la izquierda cuando andamos por su perímetro. Después observaremos que estas dos restricciones se pueden solventar de forma sencilla; en realidad la primera ni siquiera es necesaria. En primer lugar es fácil llevar la cuenta de dónde nos encontramos dentro del área de dibujo; para ello introduciremos dos variables `posX` y `posY`.

Iremos calculando la superficie según nos vayamos moviendo en la cuadrícula. Según se puede observar en la figura de la pista 2.26b, cada vez que nos movemos a la derecha, debemos quitar de la superficie los cuadrados que tengamos por debajo de la línea (están a la derecha de la misma), y los debemos añadir cuando vayamos hacia la izquierda (los que están a la izquierda de la línea). Obsérvese que el trazo del polígono pasa un número par de veces por todas las columnas (puede ser cero), la mitad hacia la derecha y la otra mitad hacia la izquierda compensándose así todas las sumas y restas, para dar finalmente el número de cuadrados contenidos dentro del polígono para cada columna.

En (a) sumamos los 4 cuadrados que hay por debajo de la línea; en (b) restamos los 5 cuadrados por debajo de la línea y llevamos acumulado -1; en (c) sumamos 6 y llevamos acumulado 5; y por último en (d) restamos 1 y nos queda finalmente (en esa columna) 4.

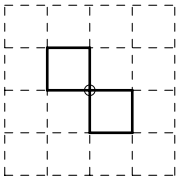
Por tanto deberemos hacer un bucle recorriendo el contorno de la figura según la entrada de datos; en cada vuelta del mismo vamos actualizando la posición, si el movimiento es hacia la derecha restamos a la superficie el contenido de la variable `posY` y si es hacia la izquierda lo sumamos.

```
string recorrido;
cout << "Dame el recorrido:  ";
cin >> recorrido;
char const baja = 'B';
char const sube = 'S';
char const izquierda = 'I';
char const derecha = 'D';
int posX = 0, posY = 0;
int superficie = 0;
for (unsigned int i = 0; i < recorrido.size(); i++) {
  switch(recorrido[i]) {
  case baja:
    posY--;
    break;
  case sube:
    posY++;
    break;
  case izquierda:
    posX--;
    superficie = superficie + posY;
    break;
  case derecha:
    posX++;
    superficie = superficie - posY;
    break;
  }
}
cout << endl << endl << "Superficie = " << superficie << endl;
```

Podemos comprobar fácilmente que en esta solución no importa si el polígono está o no por encima del punto de origen. Si estamos por debajo del punto de origen y vamos hacia la derecha deberemos añadir los cuadrados hasta el punto de origen. Pero como `posY` tendrá un valor negativo, para hacer esa suma sigue siendo válido restar el valor de esa variable. Y lo mismo ocurre si vamos hacia la izquierda.

Faltaría por último averiguar qué ocurre si, en lugar de suponer que tenemos el polígono a la izquierda, lo tenemos a la derecha. Simplemente basta con observar que, donde antes sumábamos, ahora restamos y viceversa, por lo que nos quedaría la superficie negativa. En definitiva, para que funcione en ambos casos es suficiente con devolver el valor absoluto de la superficie calculada.

Además esto implica que la superficie no se calculará de forma adecuada si la línea poligonal se cruza con ella misma, puesto que después del cruce el polígono pasa de estar a un lado a estar al otro por lo que las superficies se restarían. Así nuestro cálculo de la superficie para la siguiente figura es cero.



2 28 De notación polaca a código ensamblador

73

Numerando los registros desde el cero, llamamos `cima` al último ocupado. Inicialmente, todos están libres:

```
int cima = -1;
```

Examinando ahora la notación postfija, vemos que cada elemento (variables u operaciones) viene dado por un carácter, lo que simplifica su lectura y tratamiento:

```
int main() {
  int cima = -1;
  char c;
  cout << "Expresión postfija (acabada en punto):  ";
  while (cin >> c) {
    if ('A' <= c && c <= 'Z') {
      ⟨Lectura de una variable⟩
    } else if (c == '+') {
      ⟨Lectura y tratamiento de una suma⟩
    } else if (c == '-') {
      ⟨Lectura y tratamiento de una resta⟩
    } else if (c == '*') {
      ⟨Lectura y tratamiento de una multiplicación⟩
    } else if (c == '/') {
      ⟨Lectura y tratamiento de una división⟩
    } else if (c == '@') {
      ⟨Lectura y tratamiento de un cambio de signo⟩
    } else if (c == '.')  {
      ⟨Punto final⟩
    }
  }
}
```

Y las situaciones posibles para cada carácter leído son las siguientes:

Lectura de una variable A medida que estas variables se leen (con `LOAD`), se almacenan (con `STO`) en registros sucesivos: (`LOAD A`, `STO $0`, `LOAD B`, `STO $1`, ..., `LOAD H`, `STO $i`):

```
cima = cima + 1;
cout << "LOAD " << c << endl;
cout << "STO  $" << cima << endl;
```

Lectura y tratamiento de una operación binaria Produce que dicha operación se aplique a los contenidos de los registros penúltimo y último (`cima - 1` y `cima`), y el resultado se guarde en el penúltimo. El último registro queda libre. Por ejemplo, la suma se trata así:

```
cout << "LOAD $" << cima-1 << endl;
cout << "ADD  $" << cima << endl;
cima = cima -1;
cout << "STO  $" << cima << endl;
```

Lectura y tratamiento de un cambio de signo Esta operación se aplica al contenido del último registro (`cima`), y el resultado se guarda en el mismo:

```
cout << "LOAD $" << cima << endl;
cout << "NEG" << endl;
cout << "STO  $" << cima << endl;
```

Punto final Este carácter determina el fin de la lectura y de la generación de código ensamblador.

Así se completa el programa pedido.

2.29 Raíces y logaritmos

74 79

Raíz cuadrada Tal y como se ha dicho en el enunciado, calcular la raíz cuadrada de un real `x` equivale a buscar la solución y de la ecuación $y^2 - \mathtt{x} = 0$. Como $\mathtt{x} \geq 1$, tenemos que $1^2 - \mathtt{x} \leq 0$ y $\mathtt{x}^2 - \mathtt{x} \geq 0$; por tanto podemos establecer como límites iniciales de búsqueda `li` $= 1$ y `ls` $= x$, y luego sustituir, repetidamente, los límites izquierdo o derecho por el punto medio según convenga, cuidando de que el cero quede dentro de estos límites:

```
li = 1; ls = x;
while (ls-li >= error) {
  medio = (li+ls)/2;
  valor = medio*medio - x;
  if (valor <= 0) li = medio;
  else ls = medio;
}
raiz = ls;
```

Obsérvese que, en cada vuelta del bucle anterior, se cumple que $\mathtt{li}^2 - \mathtt{x} \leq 0 \leq \mathtt{ls}^2 - \mathtt{x}$; o sea, que el cero se mantiene entre `li` y `ls`.

El problema al considerar $0 \leq \mathtt{x} < 1$ es que $\mathtt{x}^2 - \mathtt{x} < 0$ y $1^2 - \mathtt{x} > 0$, con lo que las asignaciones iniciales de `li` y `ls`, en este caso, deben ser `li = x` y `ls = 1`.

Para que el programa sea válido para todo $\mathtt{x} \geq 0$, sólo será necesario que estas asignaciones iniciales se efectúen según el caso en que estemos:

```
if (x < 1) {
  li = x;
  ls = 1;
} else {
  li = 1;
  ls = x;
}
```

Logaritmo La inversa de la función logarítmica es la función exponencial; por tanto tendremos que calcular los ceros de la función $\mathtt{b}^y - \mathtt{x}$. Sabemos que $1 \leq \mathtt{x} \leq \mathtt{b}$ por lo que $\mathtt{b}^0 - \mathtt{x} \leq 0$ y $\mathtt{b}^1 - \mathtt{x} \geq 0$.

```
li = 0; ls = 1;
while (ls-li >= error) {
  medio = (li+ls)/2;
  valor = ⟨b^medio⟩ - x;
  if (valor <= 0) li = medio;
  else ls = medio;
}
logaritmo = ls;
```

Obsérvese que, en cada vuelta del bucle anterior, el cero se mantiene entre `li` y `ls`.

Pero, en la solución de este ejercicio, se ha prohibido usar la función exponencial. Y para solventar ese inconveniente, hemos de darnos cuenta de lo siguiente:

$$\mathtt{b}^{\mathtt{medio}} = \mathtt{b}^{(\mathtt{li}+\mathtt{ls})/2} = \sqrt{\mathtt{b}^{\mathtt{li}+\mathtt{ls}}} = \sqrt{\mathtt{b}^{\mathtt{li}}\mathtt{b}^{\mathtt{ls}}}$$

Así pues, introducimos dos variables nuevas `bLi` y `bLs`, cuyo valor en cada vuelta del bucle será $\mathtt{b}^{\mathtt{li}}$ y $\mathtt{b}^{\mathtt{ls}}$ respectivamente. Puesto que los valores iniciales de `li` y `ls` son 0 y 1, los valores iniciales de `bLi` y `bLs` serán 1 y `b`. Tampoco podemos olvidar que, al cambiar el valor de las variables `li` y `ls`, se ha de cambiar también el de `bLi` y `bLs`:

```
li = 0; ls = 1;
bLi = 1; bLs = b;
while (ls-li >= error) {
  medio = (li+ls)/2;
  bMedio = sqrt(bLi*bLs);
  valor = bMedio - x;
  if (valor <= 0) {
    li = medio;
    bLi = bMedio;
  } else {
    ls = medio;
    bLs = bMedio;
  }
}
logaritmo = ls;
```

Supongamos ahora que $\mathtt{x} > \mathtt{b}$; realizaremos entonces un bucle dividiendo `x` por `b`, de forma que en cada vuelta se verifique $\mathtt{b}^{\mathtt{exponente}} \cdot \mathtt{resto} = \mathtt{x}$. Obsérvese que, en cada momento, $\log_{\mathtt{b}}(\mathtt{x}) = \mathtt{exponente} + \log_{\mathtt{b}}(\mathtt{resto})$. Si iteramos el bucle hasta que $\mathtt{resto} \leq \mathtt{b}$, podremos aplicar el algoritmo de arriba a la variable `resto`, y puesto que $\mathtt{b} > 1$ llegará un momento que el algoritmo pare.

Si $0 < \mathtt{x} < 1$, el razonamiento es análogo, pero cambiando la división por el producto.

Obsérvese, por último, que no es necesaria una instrucción condicional para tratar ambos casos: es suficiente con un bucle a continuación del otro; si se entra en el primero no se entrará en el segundo y viceversa. En resumen, tenemos lo siguiente:

```
exponente = 0; resto = x;
while (resto > b) {
  exponente++;
  resto = resto/b;
}
while (resto < 1) {
  exponente--;
  resto = resto*b;
}
⟨logaritmo de resto⟩;
logaritmo = logaritmo + exponente;
```

2 32 Suma que te suma

75 79

Elevar al cuadrado y al cubo Deseamos calcular el cuadrado de `n` ≥ 0 sin usar multiplicaciones, sólo sumando. Nos planteamos proceder de forma incremental, calculando $\texttt{i}^2$ para $\texttt{i} = 0, 1, 2 \ldots$ hasta llegar a `n`. Una primera versión del programa sería ésta:

```
i = 0; iCuadrado = 0;
while (i < n) {
  iCuadrado = ⟨(i+1) al cuadrado⟩;
  i++;
}
```

El avance del bucle requiere calcular $(\texttt{i}+1)^2$ sin efectuar multiplicaciones. Para ello podemos utilizar el valor $\texttt{i}^2$ que en la vuelta anterior se almacenó en la variable `iCuadrado`. Como $(\texttt{i}+1)^2 = \texttt{i}^2 + 2\texttt{i} + 1$, la expresión para calcular $(\texttt{i}+1)^2$ será `iCuadrado + 2*i + 1`, quedando el programa así:

```
i = 0; iCuadrado = 0;
while (i < n) {
  iCuadrado = iCuadrado + 2*i + 1;
  i++;
}
```

Aún tenemos que eliminar la multiplicación `2*i` en cada vuelta. Podríamos utilizar que $2\texttt{i} = \texttt{i} + \texttt{i}$ y bastaría con sustituir el producto `2*i` por la suma `i+i`. Pero lo podemos hacer también de forma incremental si introducimos una variable nueva `dosIMasUno` cuyo valor en todo momento sea el de $2\texttt{i}+1$. Como en cada vuelta la variable `i` aumenta el valor en una unidad, el valor de `dosIMasUno` se actualiza aumentándolo en dos unidades. El valor inicial de `dosIMasUno` ha de ser 1, ya que `i` vale inicialmente cero:

```
i = 0; iCuadrado = 0; dosIMasUno = 1;
while (i < n) {
  iCuadrado = iCuadrado + dosIMasUno;
  dosIMasUno = ⟨2(i+1)+1⟩;
  i++;
}
```

Para acabar, basta advertir que $2(\mathtt{i}+1)+1 = 2\mathtt{i}+1+2$ con lo que finalmente el programa queda así:

```
i = 0; iCuadrado = 0; dosIMasUno = 1;
while (i < n) {
  iCuadrado = iCuadrado + dosIMasUno;
  dosIMasUno = dosIMasUno + 2;
  i++;
}
```

Calcular el cubo de un número con sumas es análogo al cálculo del cuadrado que acabamos de ver. Consideraremos una variable `iCubo` que en cada momento valga $\mathtt{i}^3$. En cada vuelta del bucle tendremos que actualizar el valor de `iCubo`. Desarrollando, tenemos lo siguiente:

$$(\mathtt{i}+1)^3 = \mathtt{i}^3 + 3\mathtt{i}^2 + 3\mathtt{i} + 1$$

Necesitamos una variable `tresICuadrado` de forma que en cada vuelta del bucle su valor sea $3\mathtt{i}^2+3\mathtt{i}+1$. Habrá que actualizar dicha variable en cada vuelta del bucle. Como antes, si desarrollamos

$$3(\mathtt{i}+1)^2 + 3(\mathtt{i}+1) + 1 = 3\mathtt{i}^2 + 3\mathtt{i} + 1 + 6\mathtt{i} + 6$$

llegamos a la asignación:

$$\mathtt{tresICuadrado} = \mathtt{tresICuadrado} + 6 * \mathtt{i} + 6$$

Ahora introducimos otra variable `seisIMasSeis` cuyo valor en cada vuelta será $6\mathtt{i}+6$. Para actualizar su valor es suficiente tener en cuenta lo siguiente:

$$6(\mathtt{i}+1) + 6 = 6\mathtt{i} + 6 + 6$$

Y juntando todas las piezas, tenemos el programa siguiente:

```
i = 0; iCubo = 0; tresICuadrado = 1; seisIMasSeis = 6;
while (i < n) {
  iCubo = iCubo + tresICuadrado;
  tresICuadrado = tresICuadrado + seisIMasSeis;
  seisIMasSeis = seisIMasSeis + 6;
  i++;
}
```

Comprobación de cuadrados y cubos perfectos En primer lugar es necesario observar que cualquier número no negativo se encuentra entre dos cuadrados perfectos:

$$\mathtt{i}^2 \leq \mathtt{n} < (\mathtt{i}+1)^2$$

El programa entonces se reducirá a encontrar el primer `i` tal que $(\mathtt{i}+1)^2 > n$.

Construiremos un bucle como en el apartado anterior: una variable contador `i`, se incrementará hasta encontrar el primer valor tal que $(\mathtt{i}+1)^2 > \mathtt{n}$; calcularemos $\mathtt{i}^2$ de forma incremental guardando su valor en `iCuadrado`. El número será un cuadrado perfecto si $\mathtt{n} = \mathtt{iCuadrado}$. Teniendo en cuenta que $(\mathtt{i}+1)^2 = \mathtt{iCuadrado} + \mathtt{dosIMasUno}$, el programa queda así:

```
i = 0; iCuadrado = 0; dosIMasUno = 1;
while ((iCuadrado + dosIMasUno) <= n) {
  iCuadrado = iCuadrado + dosIMasUno;
  dosIMasUno = dosIMasUno + 2;
  i++;
}
esCuadrado = (n == iCuadrado);
```

Si además introducimos una variable `iMasUnoCuadrado` que en todo momento valga $(\texttt{i}+1)^2$, el programa queda así:

```
i = 0; iCuadrado = 0; dosIMasUno = 1; iMasUnoCuadrado = 1;
while (iMasUnoCuadrado <= n) {
  iCuadrado = iMasUnoCuadrado;
  dosIMasUno = dosIMasUno + 2;
  iMasUnoCuadrado = iCuadrado + dosIMasUno;
  i++;
}
esCuadrado = (n == iCuadrado);
```

Para averiguar si un número es un cubo perfecto se procedería de forma similar.

Raíces cuadrada y cúbica: su parte entera En primer lugar es necesario precisar qué entendemos por parte entera de un real: el mayor entero menor o igual que el real dado. Así pues, la parte entera de la raíz cuadrada de un entero no negativo `n` es el único entero `raiz` tal que

$$\texttt{raiz} \leq \sqrt{n} < \texttt{raiz} + 1$$

Si elevamos todo al cuadrado, y sabiendo que todos son números no negativos, tenemos lo siguiente:

$$\texttt{raiz}^2 \leq n < (\texttt{raiz} + 1)^2$$

Así pues, el programa será el mismo que el del apartado anterior y la solución será el último valor de `i`:

```
i = 0; iCuadrado = 0; dosIMasUno = 1; iMasUnoCuadrado = 1;
while (iMasUnoCuadrado <= n) {
  iCuadrado = iMasUnoCuadrado;
  dosIMasUno = dosIMasUno + 2;
  iMasUnoCuadrado = iCuadrado + dosIMasUno;
  i++;
}
raiz = i;
```

La raíz cúbica se calcularía de forma similar.