Números romanos

Se muestran a continuación las cabeceras y especificaciones de un conjunto de funciones que pueden resolver el problema planteado por el proyecto de desarrollo de dos programas C++ en las clases de teoría de esta semana. Algunas de ellas han surgido de forma general al abordar el diseño de los programas en la pizarra, otras podrían surgir al comenzar a escribir el código aplicando una metodología descendente. En cualquier caso, las funciones que se presentan a continuación tienen ya parámetros, tipos de valores devueltos y especificaciones concretas.

Se sugiere implementarlas de forma descendente; es decir, comenzando por las funciones más generales y de mayor nivel de abstracción (las funciones `main` de los dos programas), hasta las más simples. El diseño de una función debe apoyarse en las de un nivel de abstracción menor, así que conviene leer detenidamente las especificaciones de las funciones que pueden facilitar el diseño de una determinada función para poder utilizarlas correctamente.

Primer nivel de abstracción

Función `main` del módulo `CalculadoraEnteros`

```
/*
 * Pre:  ---
 * Post: Ha mantenido un diálogo iterativo con el operador en el que se le ha
 *       solicitado que escribiera expresiones con romanos. Cuando la expresión
 *       era correcta, la ha vuelto a escribir junto con su resultado. Cuando
 *       el operador de la expresión era incorrecto, ha informado del error. Ha
 *       terminado cuando el operador ha escrito la orden FIN.
 */
int main();
```

Para el diseño de esta función, apóyate en la función `procesarExpresion` del segundo nivel de abstracción.

Función `main` del módulo `ProcesadorExpresiones`

```
/*
 * Pre:  ---
 * Post: Ha soliciado al operador el nombre de dos ficheros de texto. El primero
 *       representa el nombre de un fichero de texto que contiene expresiones
 *       con números romanos, a razón de una expresión por línea.
 *       El segundo representa el nombre de un fichero de
 *       texto en el que, por cada expresión correta, se ha escrito una línea
 *       con la misma expresión y su resultado. Por cada expresión incorrecta,
 *       se ha escrito una línea con un mensaje de error. El programa ha
 *       informado escribiendo en la pantalla de cuántas expresiones con romanos
 *       se han procesado, cuántas eran correctas y cuántas, incorrectas.
 */
int main();
```

Para el diseño de esta función, apóyate en las funciones `procesarExpresiones` y `procesarExpresion` del segundo nivel de abstracción.

Segundo nivel de abstracción

Función procesarExpresiones

```
/*
 * Pre:  «nombreFicheroProcesar» es el nombre de un fichero de texto que
 *       contiene expresiones con números romanos, a razón de una expresión por
 *       línea.
 * Post: Ha creado un fichero de texto denominado «nombreFicheroResultados» en
 *       el que, por cada expresión correta leída del fichero de nombre
 *       «nombreFicheroProcesar», se ha escrito una línea
 *       con la misma expresión y su resultado. Por cada expresión incorrecta,
 *       se ha escrito una línea con un mensaje de error. Ha asignado al
 *       parámetro «totalExpresiones» el número total de expresiones con romanos
 *       que se han procesado, al parámetro «totalCorrectas» el número de
 *       expresiones correctas y, al parámetro «totalIncorrectas», el de
 *       incorrectas.
 */
void procesarExpresiones(const char nombreFicheroProcesar[],
                         const char nombreFicheroResultados[],
                         int& totalCorrectas, int& totalIncorrectas)
```

Para el diseño de esta función, apóyate en la función procesarExpresion del segundo nivel de abstracción.

Función procesarExpresion

```
/*
 * Pre:  «entrada» está abierto para lectura y «salida» está abierto para
 *       escritura.
 * Post: Ha intentado leer de «entrada» una expresión entre números romanos. Si lo
 *       ha conseguido y la expresión era correcta, ha escrito en «salida» la
 *       expresión y el resultado, ha asignado a «correcta» el valor «true» y a
 *       «seguir», el valor «true». Si lo ha conseguido pero la expresión no era
 *       correcta, ha escrito en «salida» un mensaje de error, ha asignado a
 *       «correcta» el valor «false» y a «seguir», el valor «true». Si no ha
 *       conseguido leer una expresión o si la expresión era la orden "FIN", ha
 *       asignado a «seguir» el valor «false».
 */
void procesarExpresion(istream& entrada, ostream& salida,
                       bool& correcta, bool& seguir);
```

Para el diseño de esta función, apóyate en las funciones aMayusculas y esOperadorValido, y escribirResultado del tercer nivel de abstracción y en la función valorDeRomano del cuarto nivel de abstracción.

Tercer nivel de abstracción

Función aMayusculas

```
/*
 * Pre: ---
 * Post: Ha tranformado los caracteres almacenados en cadena que eran letras
 *       minúsculas en sus mayúsculas correspondientes y ha dejado inalterados
 *       sus restantes caracteres.
 */
void aMayusculas(char cadena[]);
```

Función esOperadorValido

```
/*
 * Pre:  ---
 * Post: Ha devuelto true si y solo si «operador» representa un operador
 *       binario válido, es decir, si representa las operaciones de suma, resta,
 *       multiplicación o división.
 */
bool esOperadorValido(const char operador);
```

Función escribirResultado

```
/*
 * Pre:  «operador» representa un operador binario.
 * Post: Ha escrito en «f» la expresión representada por «valor1», «operador» y
 *       «valor2» y el resultado de la misma.
 */
void escribirResultado(ostream& f, const int valor1, const char operador,
                       const int valor2)
```

Para el diseño de esta función, apóyate en la función escribirComoRomano del cuarto nivel de abstracción.

Cuarto nivel de abstracción

Función escribirComoRomano

```
/*
 * Pre:  ---
 * Post: Ha escrito en «f» el «valor», en notación romana si está en el intervalo
 *       [1,3999], y con notación arábiga en caso contrario.
 */
void escribirComoRomano(ostream& f, const int valor);
```

Para el diseño de esta función, apóyate en la función convertirARomano del cuarto nivel de abstracción.

Función convertirARomano

```
/*
 * Pre:  ---
 * Post: Ha asignado a «romano» la secuencia de caracteres que definen un número
 *       romano equivalente a «valor» (que puede ser negativo). Si «valor» no es
 *       representable por tener un valor absoluto mayor o igual que 4000, ha
 *       asignado a «romano» el valor "***".
 */
void convertirARomano(const int valor, char romano[]);
```

Para el diseño de esta función, apóyate en la función convertirDigito del quinto nivel de abstracción.

Función valorDeRomano

```
/*
 * Pre:  «romano» corresponde a un número romano correcto (que podría ser negativo).
 * Post: Ha devuelto el valor entero equivalente al número romano representado por
 *       «romano».
 */
int valorDeRomano(const char romano[]);
```

Para el diseño de esta función, apóyate en la función valorDigito del quinto nivel de abstracción.

Quinto nivel de abstracción

Función convertirDigito

```
/*
 * Pre:  «digito» >= 0 y «digito» <= 9.
 *       «uno» es una cifra romana que representa una unidad, decena, centena o
 *       millar, «cinco» es la cifra romana que representa cinco veces «uno» y
 *       «diez» es la cifra romana que representa diez veces «uno».
 *       Es decir, (uno, cinco, diez) = ('I', 'V', 'X') o ('X', 'L', 'C') o
 *       ('C', 'D', 'M') o ('M', '*', '*').
 * Post: Ha asignado a la tabla «romano» los caracteres que
 *       representan «digito» expresado en notación romana.
 */
void convertirDigito(const int digito,
                     const char uno, const char cinco, const char diez,
                     char romano[]);
```

Función valorDigito

```
/*
 * Pre:  «cifraRomana» es una cifra romana: 'I', 'V', 'X', 'L', 'C', 'D' o 'M'.
 * Post: Ha devuelto el valor entero de «cifraRomana».
 */
int valorDigito(const char digitoRomano);
```