1. **(2 puntos (sobre 7))**
Defina una clase para trabajar con una secuencia de caracteres y proporcione métodos para obtener un valor dado, obtener la longitud actual y añadir uno nuevo.

Dada una secuencia de caracteres que contiene un mensaje cifrado, se pide definir un método para construir otra secuencia nueva con el mensaje descifrado. La forma de descifrado consiste en coger la primera y última letra de cada palabra. Las palabras están separadas por uno o más espacios en blanco o el final de la secuencia. Si la última palabra no tiene espacios en blanco a su derecha, se coge sólo el primer carácter.

Por ejemplo, si denotamos el inicio y final de la secuencia con un corchete, entonces:

`[  Hidrógeno limpia  ]` se descodificaría como `Hola`

`[Hidrógeno limpia   ]` se descodificaría como `Hola`

`[Hidrógeno limpia]` se descodificaría como `Hol`

`[Hidrógeno]` se descodificaría como `H`

`[Hidrógeno   ]` se descodificaría como `Ho`

`[H]` se descodificaría como `H`

`[H   ]` se descodificaría como `H`

No hace falta incluir el programa principal.

Se supone la disponibilidad de la clase `SecuenciaCaracteres` con los métodos:

- `SecuenciaCaracteres ()`
- `int TotalUtilizados ()`
- `int Capacidad ()`
- `void Aniade (char nuevo)`
- `char Elemento (int indice)`

Se proponen dos soluciones alternativas.

**Solución 1:**

Una vez se han saltado los espacios iniciales, se procesa una palabra y se saltan los espacios que le siguen hasta alcanzar la siguiente palabra.

```
/*********************************************************************/

SecuenciaCaracteres Descifra ()
{

        SecuenciaCaracteres resultado;

        int pos = 0;
        int pos_init, pos_end;


        // Saltar espacios iniciales, si los hubiera

        while ((pos<total_utilizados) && isspace(vector_privado[pos])) pos++;
```

```
	// Mientras queden palabras que procesar

	while (pos < total_utilizados) {

		// pos_init indica el inicio de una palabra
		pos_init = pos;

		// El primer carácter de una palabra SIEMPRE se añade

		resultado.Aniade (vector_privado[pos_init]);


		// Saltar todos los caracteres de la palabra

		while (!isspace(vector_privado[pos]) && (pos<total_utilizados))
			pos++;

		// En pos hay un espacio, o se ha procesado toda la secuencia

		// Si hay algún espacio al final, se añade la última letra de la
		// palabra. Saltamos todos los espacios que hubiera para colocarse
		// al principio de la siguiente palabra para la nueva iteración.

		if (pos < total_utilizados) { // Hay espacios finales

			pos_end = pos-1;

			// Añadir la letra final si la palabra tiene más de una letra

			if (pos_end-pos_init>=1) {

				resultado.Aniade (vector_privado[pos_end]);
			}

			// Saltar los demás espacios que pudiera haber
			while (isspace(vector_privado[pos]) && (pos<total_utilizados))
				pos++;
		}

	} // while (pos < total_utilizados)

	return (resultado);
}

/*********************************************************************/
```

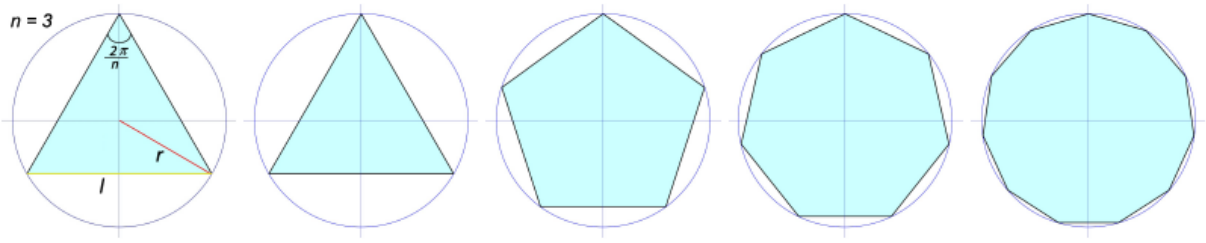
**Solución 2:**

Se analiza, carácter a carácter la secuencia, guardando el carácter anterior.

```
/*********************************************************************/

SecuenciaCaracteres Descifra2 ()
{

	SecuenciaCaracteres resultado;

	char anterior = ' ';

	int pos, pos_init;
```

```
        for (pos=0; pos < total_utilizados; pos++) {

                if (vector_privado[pos] != ' ') {

                        if (anterior == ' ') { // Primera letra de una palabra

                                resultado.Aniade(vector_privado[pos]);
                                pos_init = pos;
                        }
                }
                else { // vector_privado[pos] == ' '

                        // Si la longitud de la palabra es mayor que uno,
                        // añadir el último carácter de ésta.

                        if ((anterior != ' ') && (pos - pos_init > 1) )
                                resultado.Aniade(anterior);
                }

                anterior = vector_privado[pos];

        } //for (pos=0; pos < total_utilizados; pos++)

        return (resultado);

}

/*************************************************************************/
```

2. **(3 puntos (sobre 7))**
Un polígono regular de $n$ caras tiene $n$ lados de la misma longitud y todos los ángulos interiores son iguales. Su centro geométrico es el centro de la circunferencia circunscrita (la que lo envuelve). Supondremos que dos polígonos son distintos si se diferencian o bien en sus centros geométricos, o bien en el número de lados o bien en la longitud de cualquiera de ellos. Así pues, por ejemplo, no tendremos en cuenta las distintas posiciones en el plano que se podrían obtener girando el polígono sobre su centro.




Si llamamos $n$ al número de lados y $l$ la longitud de cualquiera de ellos, tenemos que:

- la longitud $r$ del radio de la circunferencia circunscrita viene definida por $r = l/(2\sin(\pi/n))$
- El área del polígono es $A = (n/2) \cdot r^2 \sin(2\pi/n)$
- Si queremos construir un polígono inscrito en la misma circunferencia, pero multiplicando por un entero $k$ el número de lados, la longitud de cada uno de los $kn$ lados viene dada por $l' = r\sqrt{2(1-\cos(2\pi/(kn)))}$.

Se quiere diseñar la clase `PoligonoRegular` para poder representar este tipo de polígonos y realizar las siguientes tareas:

- Calcular el perímetro del polígono.
- Calcular el área del polígono.
- Comprobar si un polígono es mayor que otro (considerando al área de cada uno)
- Construir un nuevo polígono que tenga la misma circunferencia circunscrita y con un número de lados que sea múltiplo del número de lados del polígono.

Debe tener los siguientes constructores:

- Un constructor sin parámetros en el que los valores a asignar por defecto sean: 3 para el número de lados (triángulo), 1 para la longitud y (0,0) para las coordenadas del centro.
- Un constructor que cree un polígono regular con una longitud y número de lados concretos y deje como centro el valor por defecto (0,0).
- Un constructor que cree un polígono regular con una longitud, número de lados y centro concretos.

Defina un programa que realice las siguientes tareas:

- Cree dos polígonos, **polígono1** con los valores por defecto y **polígono2** con 6 lados de longitud 4 y centrado en (0,0). El programa comprobará si **polígono1** es mayor que **polígono2**.
- Construya un nuevo polígono a partir de **polígono1**, con la misma circunferencia circunscrita y con el doble número de lados. El programa imprimirá en pantalla el área del nuevo polígono.
- Repita el proceso anterior generando polígonos con el doble número de lados en cada iteración, hasta que el polígono generado tenga un área *similar* a la del círculo delimitado por la circunferencia circunscrita. El cómputo del área del círculo se puede realizar directamente en el programa principal y consideraremos que las áreas son similares si no se diferencian en más de $10^{-5}$.
  El programa mostrará el número de lados del polígono que aproxima a la circunferencia.

Antes de describir la clase `PoligonoRegular` debe considerar la conveniencia de disponer de la conocida clase `Punto2D` para representar el punto central del polígono. Para dicha clase nos basta con una versión mínima en este problema:

```
////////////////////////////////////////////////////////////////////////

class Punto2D
{
private:
	// La pareja (x,y) son las coordenadas de un punto en un espacio 2D
	double x = 0;
	double y = 0;

public:
	/*********************************************************************/
	// Constructor sin argumentos.
	Punto2D (void) { }

	/*********************************************************************/
	// Constructor con argumentos.
	// Recibe: abscisaPunto y ordenadaPunto, la abscisa y ordenada,
	// respectivamente del punto que se está creando.

	Punto2D (double abscisaPunto, double ordenadaPunto)
	{
		SetCoordenadas (abscisaPunto, ordenadaPunto);
	}
	/*********************************************************************/
	// Método Set para fijar simultaneamente las coordenadas.
	// Recibe: abscisaPunto y ordenadaPunto, la abscisa y ordenada,
	// respectivamente del punto que se está modificando.

	void SetCoordenadas (double abscisaPunto, double ordenadaPunto)
	{
		x = abscisaPunto;
		y = ordenadaPunto;
	}
	/*********************************************************************/
};

////////////////////////////////////////////////////////////////////////
```

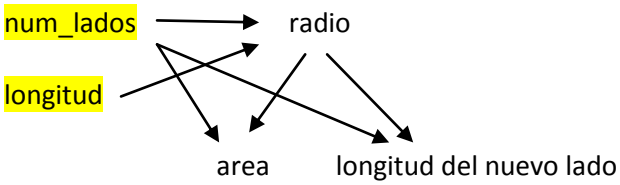
Sobre la representación de los objetos de la clase `PoligonoRegular` hay que tener cuidado de evitar la **redundancia** entre los datos miembros. Veamos:

- El **centro** del polígono debe ser un dato miembro de la clase
- El **número de lados** y la **longitud** de éstos, también. Manteniendo el número de lados fijo y variando la longitud se modifica el **radio** de la circunferencia circunscrita por lo que el radio es redundante y puede calcularse con un método.

```
class PoligonoRegular
{

private:

       Punto2D centro;      // Centro de la circunferencia circunscrita
                            // que encierra al polígono.

       int    num_lados;    // Num. de lados del polígono
       double longitud;     // Longitud de cada lado
```

Con esta representación puede calcularse:

- el **radio** a partir de la longitud y del número de lados,
- el **área** a partir del número de lados y del radio
- la longitud del lado del nuevo polígono a partir del radio y del número de lados




Constructores:

```
       /*********************************************************************/
       PoligonoRegular () : centro (Punto2D(0.0,0.0)),
                            num_lados(3),
                            longitud(1)
       { }
       /*********************************************************************/

       PoligonoRegular (int el_numero_lados, double la_longitud_lado)
                                : centro (Punto2D(0.0,0.0)),
                                  num_lados(el_numero_lados),
                                  longitud(la_longitud_lado)
       { }
       /*********************************************************************/

       PoligonoRegular(int el_numero_lados, double la_longitud_lado,
                               Punto2D el_centro)
                             : centro (el_centro),
                               num_lados(el_numero_lados),
                               longitud(la_longitud_lado)
       { }
      /*********************************************************************/
```

Métodos simples de la clase

```
/*********************************************************************/
// Métodos de consulta directa
/*********************************************************************/

Punto2D GetCentro ()
{
	return (centro);
}
/*********************************************************************/

double GetLongitud ()
{
	return (longitud);
}

/*********************************************************************/

int GetNumLados ()
{
	return (num_lados);
}

/*********************************************************************/
// Método de consulta indirecta
/*********************************************************************/

double GetRadio ()
{
	return (longitud/(2.0*sin(PI/num_lados)));
}

/*********************************************************************/
// Métodos de cálculo
/*********************************************************************/

double Perimetro ()
{
	return (longitud*num_lados);
}

/*********************************************************************/

double Area ()
{
	// Método 1:
	//return (0.25*num_lados*longitud*longitud / tan(PI/num_lados));

	// Método 2:
	double radio = GetRadio ();
	return (0.5*num_lados*radio*radio*sin(2*PI/num_lados));

}

/*********************************************************************/

bool MayorQue (PoligonoRegular otro)
{
	return (Area() > otro.Area());
}

/*********************************************************************/
```

Métodos relacionados con el problema de construir un nuevo polígono:

```
/*********************************************************************/
// Develve un nuevo polígono, inscrito en la misma circunferencia,
// pero multiplicando por un entero k el número de lados.

PoligonoRegular MultiplicaNumLados (int k)
{
       PoligonoRegular nuevo (num_lados*k, NuevaLongitudLado(k), centro);
       return (nuevo);
}
```

El método `NuevaLongitudLado ()` se puede implementar **private**:

```
/*********************************************************************/
// Devuelve la longitud del lado del nuevo poligono resultado de
// multiplicar por k el núnero de lados. El radio sigue siendo el mismo
// porque el nuevo polígono está inscrito en la misma circunferencia.

double NuevaLongitudLado (int k)
{
       double radio = GetRadio();

       double nueva_longitud = radio * sqrt(2*(1-cos((2*PI)/(k*num_lados))));

       return (nueva_longitud);
}

/*********************************************************************/
```

Finalmente, en la función `main()` puede escribirse:

1) Crear dos polígonos y compararlos

```
// Constructor sin argumentos

PoligonoRegular poligono1;


// Constructor con dos argumentos

PoligonoRegular poligono2 (6, 4);


// Comparar áreas

if (poligono1.MayorQue(poligono2))
       cout << "poligono1 es mayor que poligono2" << endl;

else
       cout << "poligono2 es mayor que poligono1" << endl;
```

2) Crear un polígonos a partir de `poligono1` y con el doble número de lados:

```
PoligonoRegular p (poligono1.MultiplicaNumLados(2));

cout << "Area = " << p.Area() << endl;
```

3) Aproximar una circunferencia creando polígonos con mayor número de lados.

Necesitaremos una función que compare dos números reales con una precisión dada:

```
/*************************************************************************/
// Función global (asociada a la constante PRECISION_SON_IGUALES) que
// permite que dos números reales muy próximos sean considerados iguales.

const double PRECISION_SON_IGUALES = 1e-5; // 0.00001

bool SonIguales(double uno, double otro)
{
	return (fabs(uno-otro) <= PRECISION_SON_IGUALES);
}
```

A continuación de la creación de `p` escribimos el código que nos permite aproximar la circunferencia construyendo sucesivos polígonos:

```
	// Calcular el área de la circunferencia circunscrita (área objetivo)

	double radio = p.GetRadio();
	double area_objetivo = PI * radio * radio;

	// Presentar datos iniciales de la aproximación

	cout << endl;
	cout << "Calculando el poligono que aproxima a una circunferencia "
	     << "de radio " << radio << endl;
	cout << "Area de la circunferencia objetivo = "
	       << setw(10) << setprecision(8) << area_objetivo << endl;
	cout << endl;


	// Calcular el polígono que aproxima la circunferencia

	while (!SonIguales(p.Area(), area_objetivo)) {

		cout << "Lados = " << setw(5) << p.GetNumLados()
		     << " Area = " << setw(10) << setprecision(8) << p.Area();
		cout << " Dif  = " << setw(10) << setprecision(8)
		     << area_objetivo - p.Area() << endl;

		// Calcular nuevo poligono

		p = p.MultiplicaNumLados(2);
	 }

	// Presentar resultado

	cout << endl;
	cout << "POLIGONO RESULTADO: " << endl;
	cout << "Lados = " << setw(5) << p.GetNumLados();
	cout << " Area = " << setw(10) << setprecision(8) << p.Area();
	cout << " Dif  = " << setw(10) << setprecision(8)
	     << area_objetivo - p.Area() << endl;
	cout << endl << endl;
```

Podría verificarse que el polígono es adecuado porque su perímetro debe ser tambi´ñen muy cercano a la longitud de la circunferencia que aproxima:

```
// El perímetro del polígono que aproxima a la circunferencia
//  circunscrita debe ser similar a la longitud de dicha circunferenca.

cout << endl;
cout << "La longitud de la circunf. circunscrita es   "
       << setw(10) << setprecision(8) << 2*PI*p.GetRadio() << endl;
cout << "El perimetro del poligono que la aproxima es "
     << setw(10) << setprecision(8) << p.Perimetro() << endl;
cout << endl;
```

3. **(2 puntos (sobre 7))**
Dada la especificación de `SecuenciaEnteros`, se quiere implementar un método llamado `SeleccionDireccional` dentro de la clase `TablaRectangularEnteros` que devuelva una secuencia de enteros que contenga una copia de los elementos de la matriz dada una posición de origen y una dirección. El método recibirá una posición origen $(i, j)$ y un dirección dada por dos enteros $(dx, dy)$. La secuencia a devolver contendrá como primer elemento el valor $(i, j)$ de la matriz. Los siguientes elementos se obtendrán dando saltos $(dx, dy)$ desde la posición origen hasta alcanzar el extremo de la matriz. Hay que realizar todas las comprobaciones necesarias antes de acceder a posiciones inválidas en la matriz. Por ejemplo, dada la matriz:

$$\begin{pmatrix} 5 & 8 & 2 & 9 & 3 \\ 1 & 0 & -2 & 9 & 2 \\ 0 & 1 & 8 & -3 & 0 \\ 2 & -2 & 4 & 8 & -5 \\ -2 & 1 & 4 & 3 & 3 \end{pmatrix}$$

la selección direccional con $(i = 0, j = 0)$ y $(dx = 1, dy = 2)$ sería {5,1,4}, y con $(i = 1, j = 3)$ y $(dx = 0, dy = 1)$ sería {9,-3,8,3}.

Con estas indicaciones, y teniendo en cuenta que se va a emplear la clase `TablaRectangularEnteros`, se trata de realizar las siguientes tareas:

*a*) Defina los datos miembros de la clase `TablaRectangularEnteros`.

*b*) Implemente los métodos necesarios de `TablaRectangularEnteros`.

*c*) Implemente el método `SeleccionDireccional` de acuerdo a las indicaciones dadas.

*d*) Escriba el código de la función main que se encarga de llamar adecuadamente al método `SeleccionDireccional`, para obtener una `SecuenciaEnteros` (resultado) a partir de una `TablaRectangularEnteros` llamada original (supondremos que la tabla original se ha creado y llenado correctamente).

| Métodos de `SecuenciaEnteros` que NO hay que implementar y se pueden usar |
|---|
| `TotalUtilizados` |
| `Capacidad` |
| `Aniade` |
| `Elemento` |

La implementación de la clase `TablaRectangularEnteros` basada en una matriz rectangular es la siguiente. Se incluyen los constructores y los métodos simples :

```
////////////////////////////////////////////////////////////////////////
// Tabla rectangular de enteros

class TablaRectangularEnteros
{

private:

    static const int NUM_FILS = 50; // Filas disponibles
    static const int NUM_COLS = 40; // Columnas disponibles

    int  matriz_privada[NUM_FILS][NUM_COLS];

    // PRE: 0 <= filas_utilizadas <= NUM_FILS
    // PRE: 0 <= col_utilizadas < NUM_COLS

    int filas_utilizadas;
    int cols_utilizadas;

public:


       /*********************************************************************/
       // Constructor a)
       // Recibe "numero_de_columnas" que indica el número de columnas
       // que deben tener TODAS las filas.

       TablaRectangularEnteros (int numero_de_columnas)
              : filas_utilizadas(0),
                cols_utilizadas(numero_de_columnas)
       {
       }

       /*********************************************************************/
       // Constructor b)
       // Recibe una secuencia de enteros. El número de elementos de la
       // secencia es el valor que se usa como "col_utilizadas"

       TablaRectangularEnteros (SecuenciaEnteros primera_fila)
              : filas_utilizadas(0),
                cols_utilizadas (primera_fila.TotalUtilizados())
       {
              Aniade(primera_fila); // Actualiza "filas_utilizadas"
       }

       /*********************************************************************/
       // Método de lectura: número máximo de filas

       int CapacidadFilas (void)
       {
              return (NUM_FILS);
       }

       /*********************************************************************/
       // Método de lectura: número máximo de columnas

       int CapacidadColumnas (void)
       {
              return (NUM_COLS);
       }
```

```
	/*********************************************************************/
	// Método de lectura: número real de filas usadas

	int FilasUtilizadas (void)
	{
		return (filas_utilizadas);
	}

	/*********************************************************************/
	// Método de lectura: número real de columnas usadas

	int ColumnasUtilizadas (void)
	{
		return (cols_utilizadas);
	}

	/*********************************************************************/
	// Método de lectura: devuelve el dato que ocupa la casilla
	// de coordenadas (fila, columna)
	//
	// PRE: 0 <= fila < filas_utilizadas
	// PRE: 0 <= columna < cols_utilizadas

	int Elemento (int fila, int columna)
	{
		return (matriz_privada[fila][columna]);
	}

	/*********************************************************************/
	// Añade una fila completa. La fila se porporciona en un objeto de
	// la clase "SecuenciaEnteros".
	//
	// PRE:  fila_nueva.TotalUtilizados() = cols_utilizadas
	// PRE:  filas_utilizadas < MAX_FIL

	void Aniade (SecuenciaEnteros fila_nueva)
	{
		int numero_columnas_fila_nueva = fila_nueva.TotalUtilizados();

		if ((filas_utilizadas < NUM_FILS) &&
			(numero_columnas_fila_nueva == cols_utilizadas)) {

			for (int col = 0; col < numero_columnas_fila_nueva; col++)

				matriz_privada[filas_utilizadas][col] =
                                        fila_nueva.Elemento(col);

			filas_utilizadas++;
		}
	}

	/*********************************************************************/

};

//////////////////////////////////////////////////////////////////////////
```

El método `SeleccionDireccional` devuelve un objeto de la clase `SecuenciaEnteros`. Su implementación es muy sencilla:

```
/**********************************************************************/
// Devuelve una secuencia de enteros, como un objeto de la clase
// "SecuenciaEnteros" resultado de la selección de elementos de la tabla.
// PRE: 0 <= i < filas_utilizadas
// PRE: 0 <= j < cols_utilizadas
// PRE: 1 <= dx, dy

SecuenciaEnteros SeleccionDireccional (int i, int j, int dx, int dy)
{
	SecuenciaEnteros solucion;

	int fila = i;
	int col  = j;

	while (fila < filas_utilizadas && col < cols_utilizadas) {

		solucion.Aniade(matriz_privada[fila][col]);

		fila += dy;
		col  += dx;
	}

	return (solucion);
}

/**********************************************************************/
```

Si suponemos que disponemos de una tabla rellena, un ejemplos de uso de este método son los siguientes (son los mostrados en el enunciado del examen):

```
int num_elementos;


// Primer ejemplo: (i = 0, j = 0) y (dx = 1, dy = 2)

SecuenciaEnteros seleccion1 (matriz.SeleccionDireccional (0, 0, 1, 2));

// Mostrar secuencia

num_elementos = seleccion1.TotalUtilizados();

for (int i=0; i<num_elementos-1; i++)
	cout << seleccion1.Elemento(i) << ",";

cout << seleccion1.Elemento (num_elementos-1 << endl;


// Segundo ejemplo: (i = 1, j = 3) y (dx = 0, dy = 1)

SecuenciaEnteros seleccion2 (matriz.SeleccionDireccional (1, 3, 0, 1));

// Mostrar secuencia

num_elementos = seleccion2.TotalUtilizados();

for (int i=0; i<num_elementos-1; i++)
	cout << seleccion2.Elemento(i)<< ",";

cout << seleccion2.Elemento (num_elementos-1 << endl;
```