2018

Programación II

Parte I - v.1

Ing. David CECCHI

UTN

CONCEPTOS BÁSICOS

En todo programa que se desarrolla no solo se debe buscar cumplir con los requerimientos del momento, también se deberá pensar a futuro, es decir tener en cuenta que todo software durante la etapa de desarrollo como así también luego de finalizado, siempre es susceptible de sufrir modificaciones. Estas modificaciones pueden tener su raíz en cuestiones tales como: nuevos requerimientos o cambios a los ya existentes, mejoras en el algoritmo haciéndolo mas eficiente en el uso de los recursos (ej. tiempo, memoria), etc. Estos factores por lo tanto obligan a ampliar y/o retocar el código fuente.
Transcurrido un período considerable desde la finalización del programa, toda modificación obligará a la reinterpretación del código, el cual durante su generación resultaba prácticamente obvio para el desarrollador por encontrarse inmerso en su resolución desde un tiempo prolongado. Dicha reinterpretación aumentara en complejidad si el código había sido elaborado por otro individuo, con su propia lógica y costumbres de programación (herramientas, técnicas, etc.).
En consecuencia, en la etapa de desarrollo se buscara que el algoritmo cumpla con los siguientes requerimientos: *claridad, legibilidad y modificabilidad*. Los cuales facilitaran el entendimiento del código generado, permitiendo de esta manera reducir los tiempos de desarrollo y la generación de programas más robustos.

a) ***Claridad:*** implica que la resolución algorítmica sea sencilla, que esté correctamente estructurada, resultando un algoritmo de fácil compresión.

b) ***Legibilidad:*** significa que en la codificación se utilizaron ***nombres adecuados*** en lo que respecta a variables, funciones, procedimientos, etc., ***comentarios aclaratorios*** y una correcta ***diagramación*** del texto.

c) ***Modificabilidad:*** cualquier cambio que sea necesario incorporar en alguna de sus partes, no obligue a realizar cambios y/o controles en todo el programa, sino limitarlo solo al módulo interesado.

TIPOS DE DATOS

Tipo	Descripción	Bits	Rango de valores	Alias
SByte	Bytes con signo	8	[-128, 127]	**sbyte**
Byte	Bytes sin signo	8	[0, 255]	**byte**
Int16	Enteros cortos con signo	16	[-32.768, 32.767]	**short**
UInt16	Enteros cortos sin signo	16	[0, 65.535]	**ushort**
Int32	Enteros	32	[-2.147.483.648, 2.147.483.647]	**int**
UInt32	Enteros sin signo	32	[0, 4.294.967.295]	**uint**
Int64	Enteros largos	64	[-9.223.372.036.854.775.808, 9.223.372.036.854.775.807]	**long**
UInt64	Enteros largos sin signo	64	[0-18.446.744.073.709.551.615]	**ulong**
Single	Reales con 7 dígitos de precisión	32	$[1,5\times10^{-45} - 3,4\times10^{38}]$	**float**
Double	Reales de 15-16 dígitos de precisión	64	$[5,0\times10^{-324} - 1,7\times10^{308}]$	**double**
Decimal	Reales de 28-29 dígitos de precisión	128	$[1,0\times10^{-28} - 7,9\times10^{28}]$	**decimal**
Boolean	Valores lógicos	32	true, false	**bool**
Char	Caracteres Unicode	16	['\u0000', '\uFFFF']	**char**
String	Cadenas de caracteres	Variable	El permitido por la memoria	**string**
Object	Cualquier objeto	Variable	Cualquier objeto	**object**
DateTime	Representa un instante de tiempo	64	01/01/0001 00:00:00, 31/12/9999 23:59:59	**DateTime**

OPERADORES

Operador Matemáticos	Significado
+	Suma
-	Resta
*	Producto
/	División
%	resto (módulo)
++	Incremento
--	Decremento

Operador Comparación	Significado
>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que
==	Igual que
!=	Distinto de

Operador Lógicos	Significado
&&	AND
||	OR
!	NOT

ESTRUCTURAS DE CONTROL

Estructura	Sintaxis	Ejemplo

Condicional	if (Condicion)
	{
	  Instrucciones a ejecutar cuando la
	  condicion resulta verdadera
	}
	else
	{
	  Instrucciones a ejecutar cuando la
	  condicion resulta falsa
	}
		if (Num1 > Num2)
		{
		  Mayor = Num1;
		}
		else
		{
		  Mayor = Num2;
		}

Selección Múltiple	switch (Expresion)
	{
	  case Valor1:
	  {
	  Instrucciones a ejecutar
	  cuando la Expresion
	  coincida con Valor1
	  break;
	  }
	  case ValorN:
	  {
	  Instrucciones a ejecutar
	  cuando la Expresion
	  coincida con ValorN
	  break;
	  }
	  default:
	  {
	  Instrucciones a ejecutar
	  cuando la Expresion
	  no coincida con ningun Case
	  break;
	  }
	}
		switch (Presion)
		{
		  case 15:
		  {
		  Coef = 3;
		  break;
		  }
		  case 17:
		  {
		  Coef = 5;
		  break;
		  }
		  case 20:
		  {
		  Coef = 7;
		  break;
		  }
		  default:
		  {
		  Coef = 0;
		  break;
		  }
		}

Para	for (Inicio;Condicion;Incremento)
	{
	  Instrucciones a ejecutar en cada
	  ciclo
	}
		for (k = 1; k <= 10; k++)
		{
		  Suma = Suma + k;
		}

Mientras	while (Condicion)
	{
	  Instrucciones a ejecutar en cada
	  ciclo
	}
		while (Cuenta <= 10)
		{
		  Cuenta = Cuenta + 1;
		}

Repetir	do
	{
	  Instrucciones a ejecutar en cada
	  ciclo
	}
	while (Condicion);
		do
		{
		  Cuenta = Cuenta + 1;
		}
		while (Cuenta <= 10);

Para Cada	foreach (Tipo Objeto in Arreglo/Coleccion)
	{
	  Instrucciones a ejecutar en cada
	  ciclo
	}
		foreach (int k in vector)
		{
		  Suma = Suma + k;
		}

DECLARACIÓN DE VARIABLES

Objetivo	Sintaxis
Declarar una variable.	TipoDeDato NombreDeVariable;
Declarar un conjunto de variables de un mismo tipo de dato.	TipoDeDato NombreDeVariable1,NombreDeVariable2,NombreDeVariable3;
Declarar una variable asignandole un valor inicial.	TipoDeDato NombreDeVariable = ValorInicial;
Declarar un vector sin especificar la cantidad de filas a contener.	TipoDeDato[] NombreDeVariable;
Declarar un vector especificando la cantidad de filas a contener.	TipoDeDato[] NombreDeVariable = new TipoDeDato[CantFilas];
Declarar un vector especificando el contenido del mismo.	TipoDeDato[] NombreDeVariable = {Valor1,Valor2,Valor3};
Declarar una matriz de 2 dimensiones sin especificar la cantidad de filas y columnas a contener.	TipoDeDato[,] NombreDeVariable;
Declarar una matriz de 2 dimensiones especificando la cantidad de filas y columnas a contener.	TipoDeDato[,] NombreDeVariable = new TipoDeDato[CantFilas,CantCols];
Declarar una matriz de 2 dimensiones de 3x2 especificando el contenido de la misma.	TipoDeDato[,] NombreDeVariable = {{Valor11,Valor12},{Valor21,Valor22},{Valor31,Valor32}};

MODIFICADOR CONST

Las variables declaradas con la palabra **const** delante del tipo de datos, indican que son de sólo lectura. Es decir, constantes. Las constantes no pueden cambiar de valor, el valor que se asigne en la declaración será el que permanezca (es obligatorio asignar un valor en la declaración). Ejemplo: const float PI=3.141592.

COMENTAR LÍNEAS DE CÓDIGO

Con frecuencia es útil incorporar dentro del código fuente comentarios aclaratorios sobre determinadas situaciones a tener en cuenta, recordatorios o temporalmente deshabilitar ciertas líneas de código que no desea que sean compiladas. Para ello existen 2 alternativas:

1. Comentar una única línea: al comienzo de la misma se deben tipear dos barras inclinadas **//**
   Ejemplo:
   //es un comentario y no será compilado

2. Comentar un conjunto de líneas consecutivas (bloque): al comienzo del mismo se debe tipear **/*** y al finalizar ***/**
   Ejemplo:
   /*es un comentario de múltiples líneas
   y no será compilado*/

FUNCIONES Y PROCEDIMIENTOS

Un **modulo** es un bloque de código creado para llevar a cabo una determinada tarea. La utilización de módulos dentro del programa permitirá, como su nombre lo indica, llegar al concepto de modularidad. En todo algoritmo se busca alcanzar la modularidad, base para lograr la tan deseada modificabilidad. Un modulo correctamente desarrollado no necesita tener conocimiento sobre la lógica implementada dentro de los módulos que invoca (concepto de caja negra). Por lo que la solución total (algoritmo principal) quedará compuesta por partes (módulos) independientes, permitiendo prácticamente que cualquier modificación requerida, se logre sin retocar nada más que el módulo involucrado.

Procedimientos: Son aquellos módulos que no devuelven un valor, en su declaración se utiliza la palabra clave **void**. Un procedimiento es utilizado para llevar a cabo una tarea determinada para la cual se considera que el modulo invocante no requerirá información de la operación a realizar. Un ejemplo practico seria un procedimiento dedicado al blanqueo de pantalla.

Funciones: Son aquellos módulos que devuelven un valor, es decir en la declaración de la función se especifica el tipo de dato (int, char, float, etc) que la misma retornará al modulo invocante. El valor retornado por la función será especificado dentro de la misma a través de la palabra clave **return** seguido por la expresión a retornar, y podrá ser utilizado por el modulo invocante para alcanzar su objetivo. El valor retornado por la función será un valor único, inclusive podría llegar a ser una estructura, pero nunca un arreglo, en todo caso un puntero al mismo (concepto que se detallara mas adelante). Un ejemplo práctico es una función dedicada a la obtención del valor absoluto de un número dado.

Procedimiento	

Declaracion	void NombreProcedimiento (TipoParam1 NombreParam1, TipoParam2 NombreParam2,….)
	{
	declaracion variables;
	codigo
	}

Procedimiento que presenta un mensaje por pantalla cuyo destinatario debe ser especificado.	

Declaracion	static void Saludar(string Usuario)
	{
	Console.WriteLine("Hola {0}.", Usuario);
	}

Utilizacion	Saludar("Pedro"); Luego de la ejecucion de esta linea, se presentará el siguiente mensaje: Hola Pedro.

Funcion	

Declaracion	TipoDatoRetornar NombreFuncion (TipoParam1 NombreParam1, TipoParam2 NombreParam2,….)
	{
	declaracion variables;
	codigo
	return (expresion);
	}

Funcion que retorna el Valor Absoluto de un número entero.	

Declaracion	static int ValorAbsoluto(int Numero)
	{
	if (Numero < 0)
	{
	return (-Numero);
	}
	else
	{
	return (Numero);
	}
	}

Utilizacion	ValorAbs = ValorAbsoluto(-3); Luego de la ejecucion de esta linea, la variable ValorAbs contendrá el valor 3.

PARÁMETROS

Un **parámetro o argumento**, es el elemento por medio del cual se transmiten datos de un módulo a otro.
Los parámetros pueden ser clasificados como:

Por Valor: también conocido como parámetro de solo entrada. Es un parámetro que, en el momento de la invocación del modulo (función/procedimiento), el valor por él recibido es copiado al mismo y por lo tanto no se tiene referencia de la variable utilizada en ese instante. En consecuencia cualquier modificación en el contenido del parámetro no se vera reflejada en la variable utilizada al momento de la invocación.

Por Referencia: es un parámetro que, en el momento de la invocación del modulo (función/procedimiento), establece una referencia con la variable utilizada en ese instante. En consecuencia cualquier modificación en el contenido del parámetro se vera reflejada en la variable utilizada al momento de la invocación. Se debe ser cuidadoso al momento de utilizar este tipo de parámetros ya que cualquier modificación en el contenido de éstos afectara al modulo invocante.
Los parámetros por referencia pueden ser a su vez clasificados como parámetros de:
a. Entrada/Salida: Para identificar un parámetro como parámetro de Entrada/Salida, éste deberá estar precedido por la sigla "**ref**" en la declaración e invocación del modulo. Los parámetros tipo arreglo son tratados como parámetros de Entrada/Salida.
b. Salida: Para identificar un parámetro como parámetro de Salida, éste deberá estar precedido por la sigla "**out**" en la declaración e invocación del modulo.

Procedimiento que intercambia el contenido de dos parametros pasados por referencia.	

Declaracion	static void Intercambiar (ref int Param1, ref int Param2)
	{
	int aux;
	aux = Param1;
	Param1 = Param2;
	Param2 = aux;
	}

Utilizacion	int Valor1, Valor2; Valor1 = 10; Valor2 = 20; Intercambiar(ref Valor1, ref Valor2); Luego de la ejecucion de esta ultima linea, la variable Valor1 contendra el valor 20, y Valor2 contendrá el valor 10.

FUNCIONES RECURSIVAS

Un tipo especial de funciones son las llamadas ***funciones recursivas***. Estas se caracterizan por, como su nombre lo indica, invocarse así mismas. Son funciones poco frecuentes y utilizadas en situaciones, algoritmos muy particulares. Se debe tener en cuenta que este tipo de funciones, por su propia naturaleza, deben contar con una condición de finalización ya que de lo contrario el modulo se invocaría por tiempo indefinido, originando un loop infinito. El hilo de ejecución en cada invocación de la función, no va a continuar dentro de la misma hasta que finalice la ejecución de la función invocada. Este proceso continuará hasta que se alcance la condición de finalización. Una vez alcanzado este punto de corte, el hilo de ejecución ira retornando de la función invocada a la invocante hasta llegar a la primera invocación. Obteniendo así el resultado buscado.

Funcion que calcula el factorial de un numero dado, en forma recursiva.	

Declaracion	static uint factorial(uint num)
	{
	if (num>1)
	{
	return num * factorial(num-1);
	}
	else
	{
	return 1;
	}
	}
	
	//tener en cuenta que los numeros negativos no tienen factorial

Utilizacion	resultado = factorial(4); Luego de la ejecucion de esta linea, la variable resultado contendra el valor 24.
Funcionamiento	*Llamada a la funcion factorial (4) desde main:* return 4 * factorial (3); **esta expresion sera resuelta solo cuando se conozca factorial(3)** *Llamada a la funcion factorial (3) desde factorial(4):* return 3 * factorial (2); **esta expresion sera resuelta solo cuando se conozca factorial(2)** *Llamada a la funcion factorial (2) desde factorial(3):* return 2 * factorial (1); **esta expresion sera resuelta solo cuando se conozca factorial(1)** *Llamada a la funcion factorial (1) desde factorial(2):* return 1;

MAIN CON PARÁMETROS

El resultado de toda compilación exitosa es un archivo con el mismo nombre que el programa fuente pero con extensión .exe (NombrePrograma.exe). De esta forma es posible ejecutar el programa desarrollado directamente desde la línea de comandos, con el solo hecho de copiar el nuevo archivo a una pc el usuario de la misma esta en condiciones de utilizarlo, sin necesidad de contar con un compilador.
Main, como su nombre lo indica, es el modulo principal del programa y como todo procedimiento/función puede aceptar parámetros que en su caso provienen de la línea de comandos. De esta manera el usuario podrá especificarle al archivo .exe ciertos elementos que deberá tener en cuenta a lo largo de su ejecución.
Para permitir que estas especificaciones brindadas por el usuario puedan ser interpretadas por el programa se deberá incorporar un arreglo denominado **args** de una dimensión de tipo string como parámetro en la definición del Main. Este parámetro se cargará automáticamente al iniciarse la ejecución del programa con los valores correspondientes.

Ejemplo

```
//De cada argumento se listara su posicion y el texto tipeado por el usuario
static void Main(string[] args)
{
   int i;
   for (i = 0; i < args.Length; i++)
   {
      Console.WriteLine("El argumento {0} es: {1}", i, args[i]);
   }
   Console.ReadKey();
}
```

Cabe destacar que C# cuenta con la posibilidad de enviar argumentos al main que serán tenidos en cuenta durante la depuración. Para ello se seguirán los siguientes pasos: Proyecto > Propiedades de … > Depurar, en la ficha presentada se visualizará un cuadro de texto descripto como “Argumentos de la línea de comandos:” en el cual se tipeará cada uno de los argumentos que se desean enviar separados por un espacio.

ARCHIVOS

En la actualidad, un programa dedicado en forma exclusiva al manejo de variables, incapaz de registrar y recuperar información generada por si mismo a lo largo de diferentes corridas, disminuye considerablemente su calidad: imposibilidad de compartir la información con otros sistemas, obligación de ejecutar nuevamente los procesos para obtener resultados alcanzados en ejecuciones previas ocasionando una importante perdida de tiempo, etc.
Por ello, el resguardo de la información es crucial en todo software y el medio de implementarlo es a través de archivos ya sean planos o bases de datos (tema no tratado en este curso).
Los archivos pueden ser clasificados según el tipo de acceso y formato de la información almacenada.

CLASIFICACIÓN POR TIPO DE ACCESO

1. Archivos Secuenciales: En los archivos secuenciales, para obtener una línea específica del archivo, se deberán leer todas las líneas anteriores a ella. Esto se debe a que en este tipo de archivos no es posible acceder a un dato (línea) específico en forma directa, en consecuencia la escritura de datos siempre se realiza al final del archivo**.**

2. Archivos Directos: Los archivos de acceso directo (aleatorio), permiten almacenar en ellos registros que poseen una estructura establecida manteniendo el tipo de dato de cada uno de sus campos. Similar al concepto de arreglos de estructuras, tema tratado en la materia Laboratorio I. La gran diferencia reside, como se puede apreciar, en que la información almacenada en un archivo no se pierde al finalizar la ejecución del programa, como si ocurre al manejar arreglos en memoria.
La característica de reconocer los tipos de datos almacenados, es posible gracias a que la información almacenada en este tipo de archivos se encuentra en formato binario.
La principal cualidad de los archivos directos, es que permiten acceder a un registro determinado sin necesidad de leer todos los anteriores.
Se debe destacar que:

- Toda operación de lectura o escritura desplazara el puntero del archivo a la próxima posición.
- Será posible ubicar al puntero en una posición determinada gracias a una función de desplazamiento.
- Para el borrado de un registro determinado se pude recurrir a una de las siguientes alternativas: marcar el registro como eliminado (dando un determinado valor al mismo dentro de un campo reservado para tal fin) y de esta manera omitirlo durante la generación de los listados y consultas. Otro camino es copiando todos los registros del archivo, excepto el registro en cuestión, para luego eliminar el archivo original y renombrar el nuevo archivo generado.

CLASIFICACIÓN POR FORMATO DE ALMACENAMIENTO

1. Archivos de Texto: En memoria el almacenamiento de cada caracter requiere 2 bytes (16 bits), el mismo espacio requerido para almacenar un entero corto (short). En los archivos de texto cada caracter es almacenado individualmente por lo que pueden ser leídos, interpretados, por cualquier procesador de texto. De esta forma, si en un archivo se desea almacenar la palabra “peras” ésta ocupará 10 bytes (cada caracter ocupa 2 bytes), lo mismo ocurre al tratar de almacenar el numero entero 15789 el cual también ocupara 10 bytes ya que se lo almacena como un conjunto de caracteres y no como el valor que realmente representa.

Manipulación de un Archivo de Texto Secuencial

```
//incluir el namespace System.IO
static void Main(string[] args)
{
FileStream Archivo;
StreamWriter StreamGrabar;
StreamReader StreamLeer;
string Cadena;
char Opcion;

Console.WriteLine("Ingrese Opcion (1-Crear 2-Agregar 3-Listar 4-Salir):");
Opcion = Console.ReadKey().KeyChar;
```

```
switch (Opcion)
{
   case '1':
      {
         Console.WriteLine("\nIngrese la primer linea a almacenar en el nuevo archivo.");
         Cadena = Console.ReadLine();
         Archivo = new FileStream("Lista.txt",FileMode.Create);
         StreamGrabar = new StreamWriter(Archivo);
         StreamGrabar.WriteLine(Cadena);
         StreamGrabar.Close();
         Archivo.Close();
         break;
      }
   case '2':
      {
         Console.WriteLine("\nIngrese una nueva linea a agregar en el archivo.");
         Cadena = Console.ReadLine();
         Archivo = new FileStream("Lista.txt",FileMode.Append);
         StreamGrabar = new StreamWriter(Archivo);
         StreamGrabar.WriteLine(Cadena);
         StreamGrabar.Close();
         Archivo.Close();
         break;
      }
   case '3':
      {
         if (File.Exists("Lista.txt"))
         {
            Console.WriteLine("\nContenido del Archivo:");
            Archivo = new FileStream("Lista.txt", FileMode.Open);
            StreamLeer = new StreamReader(Archivo);
            while (StreamLeer.EndOfStream == false)
            {
               Cadena = StreamLeer.ReadLine();
               Console.WriteLine(Cadena);
            }
            StreamLeer.Close();
            Archivo.Close();
         }
         else
         {
            Console.WriteLine("\nEl archivo no existe.");
         }
         break;
      }
}
Console.ReadKey();
}
```

2. Archivos Binarios: En ciertas ocasiones puede ser útil o imprescindible almacenar directamente el contenido de la memoria en un archivo. A este tipo de archivos se los denomina binarios, en ellos, al almacenar la palabra “peras” se ocuparán 10 bytes (2 bytes por caracter) pero al almacenar el número entero corto (short) 15789 solo se ocuparán 2 bytes. El “inconveniente” con estos archivos es que no son legibles directamente desde un procesador de texto.

Por esta razón el tipo de archivo a utilizar depende de la información a almacenar y su finalidad. Si se desea un archivo de uso general, que pueda ser leído por cualquier usuario por medio de un procesador de texto se recurrirá a un archivo de texto, en

cambio, si se desea almacenar una imagen, una estructura con diferentes tipos de datos, etc. se optara por un archivo binario.

Manipulación de un Archivo Binario Secuencial

```
//incluir el namespace System.IO
static void Main(string[] args)
{
   FileStream Archivo;
   BinaryWriter BinarioGrabar;
   BinaryReader BinarioLeer;
   string Cadena;
   int NDoc;
   string Apellido;
   string Nombre;
   float Sueldo;
   char Opcion;

   Console.WriteLine("Ingrese Opcion (1-Crear 2-Agregar 3-Listar 4-Salir):");
   Opcion = Console.ReadKey().KeyChar;

   switch (Opcion)
   {
      case '1':
         {
            Archivo = new FileStream("Lista.txt", FileMode.Create);
            BinarioGrabar = new BinaryWriter(Archivo);

            Console.WriteLine("\nIngrese el Nro Doc.:");
            Cadena = Console.ReadLine();
            NDoc = int.Parse(Cadena);
            BinarioGrabar.Write(NDoc);
            Console.WriteLine("\nIngrese el Apellido:");
            Apellido = Console.ReadLine();
            BinarioGrabar.Write(Apellido);
            Console.WriteLine("\nIngrese el Nombre:");
            Nombre = Console.ReadLine();
            BinarioGrabar.Write(Nombre);
            Console.WriteLine("\nIngrese el Sueldo:");
            Cadena = Console.ReadLine();
            Sueldo = float.Parse(Cadena);
            BinarioGrabar.Write(Sueldo);

            BinarioGrabar.Close();
            Archivo.Close();
            break;
         }
      case '2':
         {
            Archivo = new FileStream("Lista.txt", FileMode.Append);
            BinarioGrabar = new BinaryWriter(Archivo);

            Console.WriteLine("\nIngrese el Nro Doc.:");
            Cadena = Console.ReadLine();
            NDoc = int.Parse(Cadena);
            BinarioGrabar.Write(NDoc);
            Console.WriteLine("\nIngrese el Apellido:");
            Apellido = Console.ReadLine();
            BinarioGrabar.Write(Apellido);
```

```
            Console.WriteLine("\nIngrese el Nombre:");
            Nombre = Console.ReadLine();
            BinarioGrabar.Write(Nombre);
            Console.WriteLine("\nIngrese el Sueldo:");
            Cadena = Console.ReadLine();
            Sueldo = float.Parse(Cadena);
            BinarioGrabar.Write(Sueldo);

            BinarioGrabar.Close();
            Archivo.Close();
            break;
        }

      case '3':
        {
          if (File.Exists("Lista.txt"))
          {
            Console.WriteLine("\nContenido del Archivo:");
            Archivo = new FileStream("Lista.txt", FileMode.Open);
            BinarioLeer = new BinaryReader(Archivo);

            while (BinarioLeer.PeekChar() != -1)
            {
              NDoc = BinarioLeer.ReadInt32();
              Apellido = BinarioLeer.ReadString();
              Nombre = BinarioLeer.ReadString();
              Sueldo = BinarioLeer.ReadSingle();

              Console.WriteLine("{0}-{1}-{2}-{3}", NDoc, Apellido, Nombre, Sueldo);
            }
            BinarioLeer.Close();
            Archivo.Close();
          }
          else
          {
            Console.WriteLine("\nEl archivo no existe.");
          }

          break;
        }
    }
    Console.ReadKey();
}
```

PUNTEROS

A través de una tabla de direcciones, el nombre de cada variable se asocia a una dirección de memoria, en la cual se encuentra el valor contenido por la variable. Este valor deberá corresponderse con el tipo de dato especificado en la declaración de la variable. De esta forma no es necesario recordar la dirección en la que se encuentra un valor determinado, es suficiente con invocar el nombre que tiene asociado esa dirección.
Un puntero es una variable, que como todas ellas cuenta con su propia dirección en memoria pero se diferencia de las mismas en que su contenido es una dirección a otra variable.
Al momento de la declaración se deberá especificar el tipo de dato de la variable a la cual apuntará precedido por el signo "*":

```
TipoDeDato *NombreDeVariablePuntero
```

Es posible declarar punteros a punteros, estos de declaran con 2 signos "*":

```
TipoDeDato **NombreDeVariablePuntero
```

Para la manipulación de los punteros existen 2 operadores:

- **&:** permite conocer la dirección de una variable.
- ***:** permite devolver/asignar el contenido de la variable a la que apunta.

Ejemplo

```
unsafe static void Main(string[] args)
{
   float SueldoEmpleado, SueldoJefe;
   float *Importe;
   SueldoEmpleado = 1000;
   SueldoJefe = 1800;
   Importe = &SueldoEmpleado; //Importe contiene la direccion de SueldoEmpleado
   *Importe = 1150; //el empleado recibio un aumento de $150, la variable SueldoEmpleado = 1150
   Importe = &SueldoJefe; //Importe contiene la direccion de SueldoJefe
   *Importe = *Importe + 200; //el jefe recibio un aumento de $200, la variable SueldoJefe = 2000
}
```

No es posible:

- Utilizar el operador "&" con valores constantes ni expresiones.
- Modificar la dirección de una variable.
- Asignación entre punteros de diferentes tipos de datos, para realizarlo se debe efectuar previamente una conversión de tipos:

Ejemplo

```
unsafe static void Main(string[] args)
{
   long SueldoEmpleado;
   long *ImporteA;
   long *ImporteB;
   int *ImporteC;
   int **ImporteD;
   SueldoEmpleado = 1150;
   ImporteA = &SueldoEmpleado;
   ImporteB = ImporteA; //punteros del mismo tipo de dato, ImporteB apunta a la misma direccion que apunta
ImporteA
   ImporteC = (int *) ImporteA; //punteros de diferente tipo de dato, ImporteC apunta a la misma direccion que
apunta ImporteA
   ImporteC = (int *) &SueldoEmpleado;//puntero y variable de diferente tipo de dato, ImporteC apunta a la direccion
de SueldoEmpleado
   ImporteD = &ImporteC; //ImporteD apunta a la direccion que apunta ImporteC
   **ImporteD = 1350; //SueldoEmpleado=1350
}
```

Las operaciones que se pueden efectuar en los punteros son suma y resta, esto permite desplazarse hacia posiciones de memoria que se encuentren por delante y por detrás de la dirección a la que apunta. La cantidad de bytes que se desplazará por cada unidad, dependerá del tipo de dato con el que fue declarado. Esta característica permite utilizar los punteros como elementos para recorrer arreglos, entre otras alternativas.
El nombre de un arreglo es un puntero a la 1er posición del mismo.

Ejemplo

```
unsafe static void Main(string[] args)
{
float[] Sueldos;
Sueldos = new float[4];
Sueldos[0] = 1000;
Sueldos[1] = 350;
Sueldos[2] = 750;
Sueldos[3] = 950;

fixed (float* PunteroSueldo = &Sueldos[0]) //tambien podria expresarse como PunteroSueldo=Sueldos
{
   Console.WriteLine(*PunteroSueldo); // presentara por pantalla el contenido de Sueldos[0]
   Console.WriteLine(*(PunteroSueldo + 1)); // presentara por pantalla el contenido de Sueldos[1]
   Console.WriteLine(*(PunteroSueldo + 2)); // presentara por pantalla el contenido de Sueldos[2]
   Console.WriteLine(*(PunteroSueldo + 3)); // presentara por pantalla el contenido de Sueldos[3]
}

fixed (float* PunteroSueldo = &Sueldos[2])
{
   Console.WriteLine(*PunteroSueldo); // presentara por pantalla el contenido de Sueldos[2]
   Console.WriteLine(*(PunteroSueldo + 1)); // presentara por pantalla el contenido de Sueldos[3]
}

fixed (float* PunteroSueldo = &Sueldos[1], PunteroSueldo2 = &Sueldos[3])
{
   Console.WriteLine(PunteroSueldo[-1]); //presenta por pantalla el contenido de Sueldos[0]
   Console.WriteLine(PunteroSueldo[0]); //presenta por pantalla el contenido de Sueldos[1]
   Console.WriteLine(PunteroSueldo[2]); //presenta por pantalla el contenido de Sueldos[3]
   Console.WriteLine(PunteroSueldo2 - PunteroSueldo); //presenta por pantalla la cant. de elementos entre ambos (3-
1=2)
}

Console.ReadKey();
}
```

CONTROL DE ERRORES

Una excepción es un objeto derivado de System.Exception que se genera cuando en tiempo de ejecución se presenta un error y contiene información detallada del mismo.
El control de errores se realiza a través de la instrucción **try**, la cual permitirá capturar cualquier excepción presentada al intentar ejecutar el bloque de código correspondiente y brindarle un tratamiento adecuado.
Existen una cantidad considerable de clases derivadas de System.Exception que le permitirán al desarrollador tomar medidas especificas de acuerdo al tipo de error detectado, algunas de ellas son:

Excepción	Situación ante la cual se presenta
DivideByZeroException	División por cero
IndexOutOfRangeException	Índice de acceso a elemento de tabla fuera del rango válido (menor que cero o mayor que el tamaño de la tabla)
OverflowException	Desbordamiento
StackOverflowException	Desbordamiento de la pila, generalmente debido a un excesivo número de llamadas recurrentes.

Instrucción Try	

Sintaxis	`try`
	`{`
	`  //Bloque de codigo que se intentara ejecutar`
	`}`
	`catch (Excepcion1)`
	`{`
	`  //Bloque de codigo a ejecutar ante el surgimiento de un error`
	`  //de tipo Excepcion1 en el bloque try`
	`}`
	`catch (Excepcion2)`
	`{`
	`  //Bloque de codigo a ejecutar ante el surgimiento de un error`
	`  //de tipo Excepcion2 en el bloque try`
	`}`
	`.`
	`.`
	`.`
	`finally`
	`{`
	`  //Opcional:`
	`  //Bloque de codigo a ejecutar luego de los bloques try/catch`
	`}`

Ejemplo	`Try`
	`{`
	`  Console.WriteLine("Resultado es: {0}", n1/n2);`
	`}`
	`catch (DivideByZeroException E)`
	`{`
	`  Console.WriteLine("Error: {0}", E.Message);`
	`}`
	`finally`
	`{`
	`  Console.WriteLine("Operación Finalizada.");`
	`}`

GENERACIÓN DE EXCEPCIONES

Existen situaciones en las que es necesario generar una excepción propia del modulo que se este desarrollando. Para lanzar una excepción se recurre a la utilización de la instrucción ***throw***.
El siguiente ejemplo se presenta solo con fines explicativos. En el mismo se genera una excepción vinculada con la División por Cero, siendo que CSharp cuenta con una excepción para esta situación denominada DivideByZeroException.

Ejemplo

```
static void Main(string[] args)
{
   string Cadena;
   float Num1, Num2;
   char Oper;
   float Resultado;

   Console.WriteLine("Ingrese Primer Numero");
   Cadena = Console.ReadLine();
   Num1 = float.Parse(Cadena);

   Console.WriteLine("Ingrese Segundo Numero");
   Cadena = Console.ReadLine();
   Num2 = float.Parse(Cadena);

   Console.WriteLine("Ingrese Operador");
   Cadena = Console.ReadLine();
   Oper = char.Parse(Cadena);

   switch (Oper)
   {
      case '+':
         {
            Resultado = Sumar(Num1,Num2);
            Console.WriteLine("El Resultado es: {0}", Resultado);
            break;
         }
      case '-':
         {
            Resultado = Restar(Num1, Num2);
            Console.WriteLine("El Resultado es: {0}", Resultado);
            break;
         }
      case '*':
         {
            Resultado = Multiplicar(Num1, Num2);
            Console.WriteLine("El Resultado es: {0}", Resultado);
            break;
         }
      case '/':
         {
            try
            {
               Resultado = Dividir(Num1, Num2);
               Console.WriteLine("El Resultado es: {0}", Resultado);
            }
            catch (Exception e)
            {
               Console.WriteLine("Error: {0}", e.Message);
               Console.WriteLine("Por ayuda recurrir a: {0}", e.HelpLink);
```

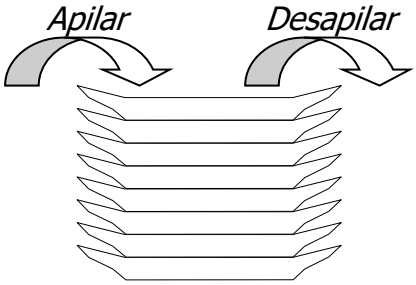
```
                Console.WriteLine("Software que genero el error: {0}", e.Source);
            }
            break;
        }
    }
    Console.ReadKey();
}


static float Sumar(float N1, float N2)
{
    return N1 + N2;
}

static float Restar(float N1, float N2)
{
    return N1 - N2;
}

static float Multiplicar(float N1, float N2)
{
    return N1 * N2;
}

static float Dividir(float N1, float N2)
{
    if (N2 != 0)
    {
        return N1 / N2;
    }
    else
    {
        Exception Excep = new Exception("Intento de division por cero.");
        Excep.HelpLink = "Consultar Fundamentos Matematicos.";
        Excep.Source = "Programa Calculadora";
        throw Excep;
    }
}
```

TIPOS DE DATOS ABSTRACTOS (TDA)

Un Tipo de Dato Abstracto (TDA) esta constituido por un conjunto de operaciones que permiten acceder y/o modificar la información por él contenida.

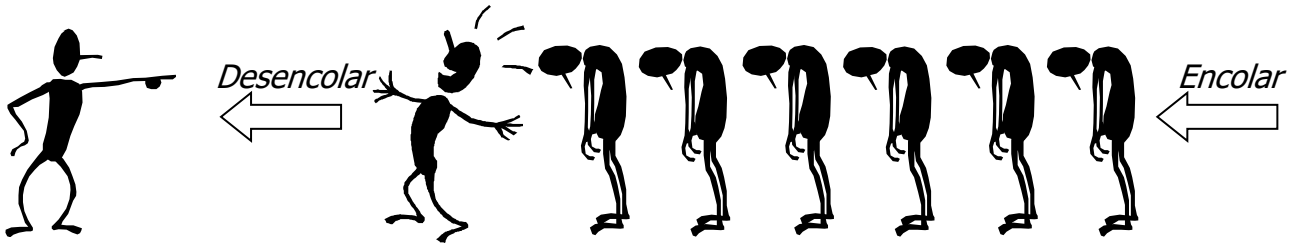
PILAS

Una Pila es una estructura de datos lineal en la cual solo se puede acceder a un único elemento de todos los almacenados, este es el que se encuentra en la *cima*. Esta forma de acceder a la información recibe el nombre de LIFO (Last In First Out): el último elemento en entrar a la pila es primer elemento en salir. Un ejemplo comúnmente utilizado para explicar el funcionamiento de una pila esta dado por una torre de platos donde a medida que los platos llegan se van colocando en la cima de la torre y de ser necesario extraer un plato, será el último apilado es decir el que se encontraba en la cima.
Por lo tanto, toda pila para su correcto funcionamiento deberá contar con una función que permita agregar (apilar) y otra quitar (desapilar) un elemento de la pila. También podrá contar con funciones que permitan saber cual es su cima, si la pila se encuentra vacía o llena.



COLAS

Una Cola es una estructura de datos lineal en la cual los elementos ingresan por un extremo y se extraen por el opuesto. Esta forma de acceder a la información recibe el nombre de FIFO (First In First Out): el primer elemento en entrar a la cola es primer elemento en salir. Un ejemplo comúnmente utilizado para explicar el funcionamiento de una cola es una fila de personas esperando ser atendidas por el cajero del banco, donde el primero en ser atendido será aquel que llego primero a la fila, es decir que se respeta el orden de llegada ya que el primero en ser atendido es el que lleva mas tiempo esperando.
Por lo tanto, este tipo de estructura debe contar con una función que permita agregar (encolar) y otra quitar (desencolar) un elemento de la cola.

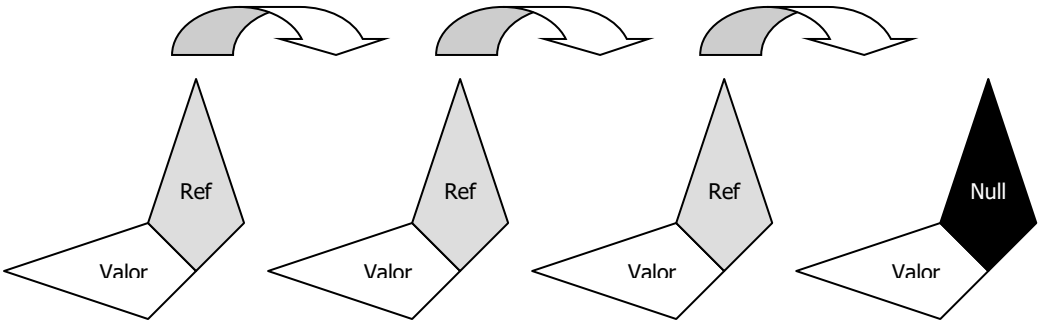


LISTAS

Una Lista es una estructura de datos lineal compuesta por nodos donde cada nodo almacena un dato y una referencia al próximo nodo de la lista. En el último nodo, la referencia al próximo de la lista contiene un null. Para insertar un nuevo nodo, se tomara el nodo inmediatamente anterior y se le modificara su referencia para que apunte al nuevo nodo, mientras que la referencia que contenía anteriormente pasara a ser la referencia contenida en el nuevo nodo. Con un criterio similar se procede ante la eliminación de un nodo, se toma el nodo inmediatamente anterior al nodo que se trata de eliminar y se modificara su referencia para que la misma apunte al nodo referenciado por el que se desea eliminar. De esta manera se mantiene la secuencialidad. Este tipo de listas son denominadas *listas simplemente enlazadas*, ya que cada nodo solo mantiene una referencia al nodo siguiente y por lo tanto pueden ser recorridas en una única dirección.
Otra clasificación son las *listas doblemente enlazadas*, donde cada nodo almacena una referencia al nodo anterior y otra al nodo siguiente. Permitiendo así recorrerla en ambas direcciones.

Una ejemplo practico del uso de listas enlazadas se encuentra en los sistemas de almacenamiento de archivos en donde para recuperar un archivo del disco se debe seguir una serie de cluster que no necesariamente se encuentran contiguos.
Existen casos en los cuales se utiliza la cabecera de la lista para almacenar más información, por ej. la cantidad de elementos que componen la lista, cual fue el último nodo visitado, etc.

Lista Simplemente Enlazada



Lista Doblemente Enlazada