Traducción Literal del libro original de Nantucket Referencias de Clipper 5.0.

Editado en Microsoft Word por: **Ivanhoe Rodriguiez Ojeda**

Marfil Negro

INDICE

PREFACIO

El presente manual ha sido elaborado a partir de la traducción literal del contenido del capítulo Conceptos Básicos del libro original de Nantucket Referencias de Clipper 5.0.

La intención de este empeño ha sido que los programadores Clipper, no desistan de la idea de utilizarlo, que sigan con esta nueva versión lo que es ya el inicio de la prometedora NFT (Nantucket Future Technologies - Tecnología de
Nantucket para el Futuro).

Se hizo incapié en éste capítulo por ser el que explica todo lo referente al nuevo modo de trabajo que trae consigo la nueva versión de Clipper 5.0 y 5.01 especialmente las declaraciones de variables, el tratamiento de arreglos, los bloques de código y los principios de programación orientada a objeto con sus cuatro clases permitidas (Error, TBrowse, TBColumn y Get). No se tradujeron los capítulos sobre las funciones y los comandos estandar por ser similares a las de versiones anteriores aunque aumenta en cantidad y en el modo de trabajo de algunas de ellas así como otras quedan obsoletas; no obstante, en buena medida ellas son tratadas en los difenrentes tópicos del capítulo, aunque las explicaciones de las mismas quedan bien especificadas en las utilidades Norton Guides que vienen incluidas en el paquete de instalación del software. Además existe un pequeño manual de Referencia Rápida al Clipper 5.0 que trae también algunas explicaciones adicionales al respecto.

Cualquier diferencia de conceptos entre el Clipper 5.0 y la versión Clipper 5.01 no están reflejadas en esta literatura. No obstante, pensamos que el principal avance de la versión 5.0 con respecto a las anteriores está en su nueva arquitectura y en las ventajas descritas anteriormente, en las cuales no encontramos ninguna diferencia en cuanto a concepto, durante el proceso de traducción en el cual se elaboraron numerosos programas de ejemplos.

El trabajo de traducción fue realizado por cinco especialistas SAD que conforman el grupo de trabajo autodenominado "Grupo Laberinto" dedicado fundamentalmente a desarrollar herramientas de extensión al lenguaje Clipper.

Otros trabajos desarrollados por este grupo son:

- Biblioteca de comunicaciones por vía TELE y RADIO para programadores de Clipper 5.01. (LC.LIB). Incluye la Norton Guides y el software LC.EXE. (ofertado)

- BackUp (BCK.EXE) para almacenar la información en el espacio disponible de los disquetes, incluye variante de compactación antes del backup. (ofertado)

- Biblioteca de propósito general para programadores de Clipper 5.01. (En proceso)

El lenguaje Clipper consiste en un lenguaje de programación estructurado y una colección de comandos standard, funciones, y objetos que le permiten crear su programa de aplicación. Este capítulo proporciona un panorama de la
construcción del lenguaje Clipper y una mejor idea del sistema. El mismo es básico para que Ud. entienda que ofrece Clipper como lenguaje de desarrollo de aplicaciones, así como, lo que representan los avances de Clipper 5.0 en la evolución de la arquitectura del lenguaje Clipper.

Cada uno de los comandos standard, funciones y sentencias referidas en este capítulo, están documentados en los subsiguientes capítulos de este libro.

1.- La estructura de un programa Clipper.

Un programa Clipper es una colección de sentencias sujetas a reglas definidas por el lenguaje Clipper. Una sentencia de programa puede tener diferentes formas, incluyendo:

- invocación de comandos

- llamadas a funciones (como son bibliotecas, pseudo-funciones, o funciones definidas por el usuario).
- llamadas a procedimientos
- directivas al pre-procesador
- estructuras de control o sentencias de declaración
- sentencias de asignación
- comentarios

Un programa es almacenado en un fichero texto con una extensión (.prg). En general, los espacios en blanco (blancos, tabulaciones) son ignorados por el compilador, permitiéndole un formato a sus programas de lectura amena. De cualquier modo, el compilador interpreta el carriage return/line feed como el fin de la sentencia. (Ver para excepciones la sección Continuación)

1.1.- Un programa típico en Clipper.

El siguiente programa ejemplo Menu.prg, es usado a lo largo de la discusión de esta sección para ilustrar varios de los componentes de un programa Clipper y dar una idea de un programa típico:

```
// Menu.prg -- Muestra menu y procesa sus opciones

#include "Database.prg" // Contiene funciones genéricas de B.D.

// Definición de constantes
#define ONE   1
#define TWO   2
#define THREE 3

// Aquí van las declaraciones de variables file-wide
PROCEDURE Main
  // Declaraciones de variables locales
LOCAL nChoice

  // Ciclo continuo que muestra el menu principal
DO WHILE .T.
    // Llamada a función que muestra menu principal
  nChoice := MainMenu()

    // Estructura del Case para llamar a funciones en
    // dependencia del valor retornado por MainMenu()
    DO CASE
      // Llamadas a funciones en la estructura Case definidas
      // en Database.prg, incluido al inicio de este fichero
      CASE nChoice = ONE
        AddRecs()
      CASE nChoice = TWO
        EditRecs()
      CASE nChoice = THREE
        Reports()
      OTHERWISE
        EXIT     // Sale fuera del ciclo
    ENDCASE
ENDDO
RETURN

FUNCTION MainMenu( menuChoice )
 CLEAR
 SET WRAP ON
 SET MESSAGE TO CENTER
```

```
@  6, 10 PROMPT "Adicionar" MESSAGE "Nueva Cuenta"
@  7, 10 PROMPT "Editar"    MESSAGE "Cambiar cuenta"
@  8, 10 PROMPT "Reportes"  MESSAGE "Imprimir reportes de cuentas"
@ 10, 10 PROMPT "Quit"      MESSAGE "Retornar al DOS"
MENU TO menuChoice
CLEAR
RETURN (menuChoice)
```

1.2.- Definición de un procedimiento principal.

Un procedimiento principal en cualquier aplicación Clipper usualmente tiene su definición en un procedimiento principal (o función) localizado al inicio del fichero programa (.prg).

En nuestro ejemplo, la definición del procedimiento principal comienza con la siguiente sentencia:

```
PROCEDURE Main
```

Esta sentencia marca el inicio del procedimiento y le da nombre al mismo. Después de compilada y redactada la aplicación, este es el procedimiento que sirve como rutina de inicio cuando la aplicación es ejecutada. En realidad Ud. debe compilar con la opción /N si su aplicación es designada como un procedimiento principal, para indicarle al compilador desde donde será generado el comienzo de la misma.

Igual que otras definiciones de procedimientos y funciones, el procedimiento principal consiste en sentencias, comandos y llamadas a funciones que ejecutan una tarea en particular. Típicamente la declaración de variables aparece al inicio del procedimiento.

Siguiendo a la declaración de variables está el cuerpo del procedimiento principal, el cual, usualmente consiste en una o más llamadas a funciones. Dependiendo de la naturaleza de la aplicación, las llamadas a funciones pueden aparecer en una estructura de control, como en Menu.prg:

```
DO WHILE .T.
   nChoice := MainMenu()
   DO CASE
     CASE nChoice = ONE
          AddRecs()
     CASE nChoice = TWO
          EditRecs()
     CASE nChoice = THREE
          Reports()
     OTHERWISE
          EXIT
   ENDCASE
ENDDO
```

Finalmente, el fin de un procedimiento principal puede ser indicado por una sentencia RETURN. La sentencia RETURN no es requerida como parte de la definición del procedimiento; pero es incluida para una lectura amena. Si la sentencia RETURN no está presente, la próxima sentencia de PROCEDURE o FUNCTION (o fin de fichero) es usada para indicar el fin de la definición anterior.

1.3.- Llamadas a funciones y procedimientos.

Dentro de una rutina, las llamadas a otras rutinas son hechas usando sus nombres. Para las funciones, el nombre debe estar seguido por sus argumentos encerrados entre paréntesis. Una llamada a función es una expresión cuyo tipo de dato es determinado por el valor que esta retorna. Por lo tanto, una llamada a función puede ser parte de otra expresión. Las llamadas a funciones son iguales independientemente de como la función

esté definida. Por ejemplo, las llamadas a funciones standards de la biblioteca Clipper son idénticas que las llamadas a funciones definidas por el usuario.

La siguiente línea de código de Menu.prg ilustra una llamada a función como parte de otra sentencia:

```
nChoice := MainMenu()
```

Los procedimientos, por otra parte, siempre retornan NIL. De este modo, una llamada a un procedimiento es normalmente hecha como una sentencia de programa independiente. Un procedimiento puede aceptar parámetros. Los parámetros son pasados hacia el procedimiento usando la misma sintáxis que una función (esto es, una lista de parámetros encerrados entre paréntesis).

De este modo, en Menu.prg AddRecs(), EditRecs() y Reports() podrán ser definidas como funciones o procedimientos sin afectar las convenciones de llamadas.

1.4.- Declaración de variables.

En Clipper las variables dinámicas pueden ser creadas e inicializadas prescindiendo de las sentencias formales de declaración. El alcance de una variable dinámica que no fue declarada, es privada para la rutina donde a variable está inicializada por primera vez. Además, Clipper es un lenguaje weak-typed (tipo débil), esto significa que el tipo de dato de una variable nunca es declarado solo es declarado el alcance o visibilidad de una variable.

El alcance de una variable debe ser declarado literalmente. En Menu.prg existe una sentencia de declaración que fija una variable local para recibir el valor de una opción de menu:

```
LOCAL nChoice
```

Esta sentencia declara nChoice como local. Excepto en el caso de la sentencia PUBLIC, la declaración de variables asigna un valor NIL como valor inicial a las mismas.

El lugar donde Ud. declara una variable en su programa, permite conocer su alcance. Por ejemplo, las declaraciones STATIC hechas antes de la primera definición de procedimiento o función tiene un alcance para todo el fichero (es decir, son variables para todas las rutinas del fichero), por el contrario, esas declaraciones hechas dentro de la definición de una rutina, son locales para esa rutina. (Ver la sección de Variables en este capítulo para más información sobre la declaración de variables.

1.5.- Definición de Procedimientos y Funciones.

Las definiciones de procedimientos y funciones son también parte de un programa típico. En Menu.prg existe solo una definición de función la cual se muestra abajo:

```
FUNCTION MainMenu(menuChoice)
  CLEAR
  SET WRAP ON
  SET MESSAGE TO CENTER
  @  6, 10 PROMPT "Adicionar" MESSAGE "New Account"
  @  7, 10 PROMPT "Editar"    MESSAGE "Change Account"
  @  8, 10 PROMPT "Reportes"  MESSAGE "Print Account Reports"
  @ 10, 10 PROMPT "Quit"      MESSAGE "Return to DOS"
  MENU TO menuChoice
  CLEAR
  RETURN ( menuChoice )
```

Esta definición es típica, consistiendo de una declaración de nombre de función con una lista de parámetros entre paréntesis, seguida por el cuerpo de la función y terminando con una sentencia de retorno pasando un valor a la rutina que la llamó. En Clipper toda definición de función debe retornar un valor.

La definición de un procedimiento es similar a la de una función, la única diferencia está en que la palabra clave PROCEDURE es usada para definir el nombre del procedimiento y sus parámetros y no es necesario retornar valor ya que los procedimientos retornan automáticamente el valor NIL.

Las rutinas que son definidas dentro de un fichero o programa son normalmente usadas solo por otras rutinas en este fichero. Las rutinas que son más generales, son usualmente almacenadas en un fichero, frecuentemente llamado fichero de procedimientos y funciones, que pueden ser compartidos por muchos programas. En Menu.prg las funciones AddRecs(), EditRecs() y Reports() son llamadas pero no definidas en el programa. Ellas están definidas en Database.prg el cual es incluido al inicio de Menu.prg.

1.6.- Directivas al pre-procesador.

Las directivas al pre-procesador pueden ser además parte de un programa Clipper. Estas directivas son instrucciones al compilador y no sentencias a compilar. Las directivas al pre- procesador siempre comienzan con el símbolo número (#). En Menu.prg diversas directivas aparecen al comienzo del fichero.
La primera es la directiva:

```
# include "Database.prg"
```

Esta línea de código lee el contenido de Database.prg hacia el fichero actual, cuando este es compilado. Entonces, todos los procedimientos y funciones definidos en él, están disponibles para su uso. En el ejemplo que se viene siguiendo, es asumido que las funciones AddRecs(), EditRecs() y Reports() están definidas en Database.prg.

A continuación aparecen varias definiciones de constantes de manera expresa. Estas directivas asignan valores constantes a nombres de identificadores que son usados a lo largo de las sentencias del programa. Donde quiera que el nombre del identificador es encontrado por el compilador, el valor asociado es sustituído en su lugar:

```
#define ONE   1
#define TWO   2
#define THREE 3
```

1.7.- Comentarios.

Los comentarios en un programa pueden aparecer en cualquier lugar, dependiendo del efecto que Ud. desee. Para comentarios largos que ocupan varias líneas o bloques de comentarios, use la siguiente forma:

```
/* El slash-asterisco inicia un bloque de comentario. Todo texto que sigue a este indicador especial de
comentario es ignorado por el compilador incluyendo el carriage return, hasta que un asterisco-slash es
encontrado,
para indicar el fin del bloque de comentarios. */
```

Para una simple línea de comentario utilice doble-slash (//) para comenzar el comentario. Todo texto que sigue a este símbolo de comentario hasta el próximo carriage return/line feed es tratado como un comentario, siendo ignorado por
el compilador. El símbolo asterisco (*) también puede ser usado para una simple línea de comentarios como en los programas dBase.

En nuestro ejemplo existen varios comentarios formulados de estas formas uno de los cuales es mostrado a continuación:

```
// Ciclo continuo que muestra el menu principal
```

Los comentarios no tienen que estar en un bloque o en una línea por si sola. El doble slash puede también ser empleado para ubicar un comentario al final de otra línea de sentencia como en el siguiente ejemplo:

```
EXIT    // Sale fuera del ciclo
```

El doble ampersand (&&) también puede ser usado para una línea de comentario. Con ambos símbolos de comentario de línea todo texto que sigue al símbolo hasta el próximo carriage return/line feed es ignorado por el compilador.

1.8.- Continuación.

Puesto que el compilador Clipper interpreta el par de caracteres carriage return/line feed como una marca de fin de una sentencia de programa, esto fuerza a que toda sentencia deba estar en una simple línea y además, solo una sentencia de programa puede estar sobre la misma línea. No obstante, el caracter punto y coma (;) puede ser usado para especificar una línea de múltiples sentencias o una sentencia de múltiples líneas.

Para continuar una sentencia en más de una línea (es decir, sentencias de multiples líneas), se pone un punto y coma al final de la primera línea antes del carriage return/line feed, y se pone el resto de la sentencia en una nueva línea. Cuando el compilador encuentra un punto y coma seguido por un carriage return/line feed, este continua leyendo la sentencia en la próxima línea. No existe en el ejemplo Menu.prg ninguna continuación de sentencia, pero los comandos PROMPTs podrían ser formateados como sigue:

```
@  6, 10 PROMPT "Adicionar" ;
  MESSAGE "Nuevo Cuenta"
@  7, 10 PROMPT "Editar" ;
  MESSAGE "Cambiar Cuenta"
@  8, 10 PROMPT "Reportes" ;
  MESSAGE "Imprimir reportes de cuentas"
@ 10, 10 PROMPT "Quit"  ;
  MESSAGE "Retornar al DOS"
```

Continuando una línea de esta forma puede ser repartida una simple sentencia en varias líneas. Usando esta característica, Ud. puede formatear una expresión muy larga para hacerla más entendible.

Para formar una línea de múltiples sentencias, simplemente se coloca un punto y coma entre las sentencias sin carriage return/line feed. Cuando el compilador encuentra algún punto y coma, asume que la próxima sentencia viene en la misma línea. Por ejemplo, la estructura CASE en Menu.prg podrá ser codificada como sigue:

```
DO CASE
  CASE nChoice = ONE   ; AddRecs()
  CASE nChoice = TWO   ; EditRecs()
  CASE nChoice = THREE ; Reports()
  OTHERWISE            ; EXIT
ENDCASE
```

1.9.- Palabras Reservadas.

Existen muchas palabras que son reservadas en Clipper 5.0 las cuales son listadas en el Apéndice Palabras Reservadas en este libro. Las palabras reservadas no pueden ser utilizadas como nombres en una sentencia de programa. Además, para estas palabras reservadas es ilegal comenzar con el caracter de subrayado (_), para usarlas como identificador.

2. Funciones y Procedimientos.

Las funciones y procedimientos son el elemento básico en la construcción de los bloques de programas Clipper. Muchos programas están hechos de uno o más procedimientos y funciones usuarios. Estas construcciones de bloques consisten de un grupo de sentencias que llaman a una simple tarea a acción. Ellas son similares a las funciones de C, Pascal u otros superlenguajes.

La visibilidad de los nombres de procedimientos y funciones cae dentro de dos clases. Las funciones y procedimientos visibles en cualquier parte del programa son referenciadas como públicas y declaradas con las sentencias FUNCTION o PROCEDURE. Las funciones y procedimientos que son visibles solo dentro del programa actual (.prg) son referenciados como estáticos y son declarados con las sentencias STATIC FUNCTION y STATIC PROCEDURE. Las funciones y procedimientos antes mencionados tienen alcance file-wide.

Los procedimientos y funciones estáticos son muy convenientes de utilizar por un número de razones. Primero, el límite de visibilidad de los nombres de un procedimiento o función restringe de este modo el acceso a los mismos. Por esta razón, subsistemas dentro de un simple programa (.prg) pueden poseer un protocolo de acceso con una serie de procedimientos y funciones públicas y ocultar los detalles de la implementación del subsistema dentro de procedimientos y funciones estáticas. Segundo ya que las referencias a procedimientos y funciones estáticas son resueltas en tiempo de compilación, ello asegura la prioridad de estas en relación a los procedimientos y funciones públicas los cuales son resueltos en tiempo de redacción. Esto asegura que dentro de un fichero programa, una referencia a un procedimiento o función estática ejecute la rutina aunque este nombre sea el mismo que el de una función pública.

2.1.- Definiendo procedimientos y funciones.

La definición de procedimientos y funciones son completamente similares teniendo cada uno una ligera diferencia en los requerimientos para el retorno de valores.

2.1.1.- Definiendo funciones de usuario.

Una función definida por el usuario puede definirse en cualquier parte de un fichero de programa (. prg), pero las definiciones no pueden estar anidadas. Una definición de función tiene la siguiente estructura básica:

```
[STATIC] FUNCTION <Identificador> [(<lista de parámetros>)]
    [<declaración de variables>]
    .
    . <sentencias ejecutables>
    .
    RETURN < expresión de retorno >
```

Una definición de función consiste de una sentencia de declaración de función con una lista opcional de declaración de parámetros. Siguiendo la declaración de la función están un conjunto de sentencias opcionales de declaración de
variables tales como LOCAL, STATIC, FIELD o MEMVAR. Como mencionamos anteriormente, una función debe retornar un valor; por tanto una sentencia RETURN con un valor de retorno, debe ser especificado en algun lugar en el cuerpo de la función. Una definición de función comienza con la sentencia de declaración FUNCTION y termina con la próxima sentencia FUNCTION o PROCEDURE o el fin de fichero.

Si la declaración de la función usuario comienza con la palabra clave STATIC, la función es visible solo por aquellos procedimientos y funciones usuarios declaradas en el mismo fichero de programa (.prg).

A continuación, un ejemplo típico de una definición de función usuario:

```
FUNCTION AmPm (cTime)
```

```
IF VAL(cTime) < 12
  cTime += " am"
ELSEIF VAL(cTime) = 12
  cTime += " pm"
ELSE
  cTime := STR( VAL( cTime) - 12, 2 ) + ;
        SUBSTR( cTime, 3 ) + " pm"
ENDIF
RETURN cTime
```

2.1.2.- Definiendo procedimientos.

Los procedimientos son idénticos a las funciones de usuario con la excepción de que no se requiere retornar valor y por tanto no es necesaria la sentencia RETURN. Al igual que las funciones usuario los procedimientos pueden ser
definidos en cualquier parte del fichero programa (.prg) pero las definiciones no pueden ser anidadas. A continuación, la forma general de las definiciones de un procedimiento:

```
[STATIC] PROCEDURE <Identificador> [(<lista de parámetros>)]
    [<declaración de variables>]
    .
    . <sentencias ejecutables>
    .

    [RETURN]
```

Parecido a una definición de función, una definición de procedimiento comienza con una declaración PROCEDURE y termina con la próxima sentencia FUNCTION o PROCEDURE o el fin de fichero.

Si la declaración del procedimiento comienza con la palabra clave STATIC, el mismo es visible solo en aquellos procedimientos y funciones de usuarios declaradas dentro del mismo fichero de programa (.prg).

2.2.- Llamando a funciones y procedimientos.

No obstante a que las funciones y procedimientos son completamente similares, ellos pueden ser llamados de diferentes formas.

2.2.1.- Llamando a una función.

Las funciones de usuario que Ud. crea y especifica, no son diferentes a las funciones standards que vienen suministradas y por tanto son llamadas en la misma forma. Las funciones pueden aparecer especificadas tanto en una expresión como en una sentencia. Por ejemplo, las siguientes llamadas a funciones son reales:

```
? "Este es el " + Ordinal( DATE()) + "dia"   // expresion
Report()                                     // sentencia
Result := Report( "Trimestralmente" )        // expresion
```

Cuando una función es llamada, esta debe estar siempre especificada incluyendo apertura y cierre de paréntesis. Si son pasados argumentos hacia la función llamada, ellos deben especificarse entre paréntesis y separados por coma. Por ejemplo:

```
? MajorFunc( "UNO", "DOS", "TRES" )
```

2.2.2.- Llamando a un procedimiento.

Un procedimiento puede ser llamado utilizando la sintaxis de llamada a una función especificando la llamada al procedimiento como una sentencia. Aqui Ud. llama al procedimiento como si fuera una función de usuario Clipper especificada como una sentencia, de esta forma:

```
<procedimiento>([<lista de parámetros>])
```

La segunda es usando el comando DO ... WITH. **Este método no es recomendado ya que pasa los argumentos por referencia implícitamente**.

2.3.- Pasando parámetros.

Cuando Ud. invoca a una función o procedimiento usuario, Ud. puede pasar valores y referencias a estos. Esta facilidad le permite a Ud. crear rutinas cajas-negras que pueden operar sobre datos independientemente de la rutina que la llamó y viceversa. En la siguiente discusión se definen varios aspectos del pase de parámetros en Clipper.

2.3.1.- Parámetros y argumentos.

El pase de parámetros desde una rutina que llama hacia una rutina que es invocada incluye dos aspectos, uno para la parte que llama y otro para la parte que recibe. En la parte que llama, el pase de valores y referencias son indicados como argumentos o parámetros actuales. Por ejemplo, la siguiente llamada a función pasa dos parámetros, una constante y una variable:

```
LOCAL nLineLength := 80
? Center( "Esta es una cadena", nLineLength )
```

En la parte que recibe las variables especificadas son referenciadas como parámetros o parámetros formales. Por ejemplo, aquí las variables son especificadas para recibir una cadena y centrarla de acuerdo a la longitud de la línea:

```
FUNCTION Center( cString, nLen )
   RETURN PADC( cString, nLen, " " )
```

Las variables especificadas para recibir, son colocadas para los valores y referencias obtenidas desde la rutina que llama. Cuando un procedimiento o función usuario es llamado, los valores y las referencias especificadas como argumentos de la rutina de invocación, son asumidos por los correspondientes parámetros que reciben en la rutina invocada. En Clipper 5.0 existen dos formas para especificar los parámetros dependiendo de la clase de almacenamiento del parámetro que Ud. quiere usar. Los parámetros especificados como una parte de la declaración de procedimiento o función son declarados parámetros y son iguales a variables locales. Ellos solo tienen visibilidad dentro de la rutina llamada y tienen el mismo tiempo de vida de la rutina. Los parámetros especificados como argumentos de la sentencia PARAMETERS, son creados como variables privadas y son ocultas para cualquier variable privada o pública o arreglos del mismo nombre proveniente desde funciones y procedimientos de un nivel superior. En versiones de Clipper anteriores a Clipper 5.0, así como en otros dialectos, esta fue la única vía de recibir parámetros.

Nota: En Clipper 5.0, Ud. no puede mezclar declaraciones de parámetros y una sentencia PARAMETERS dentro de una misma definición de función o procedimiento usuario. Este intento provocará un mensaje de error fatal del compilador.

En Clipper, los parámetros son recibidos en el orden que son pasados y el número de argumentos no tiene que ser igual que el número de parámetros especificados. En efecto, los argumentos pueden ser saltados

dentro de la lista de argumentos cuando una rutina es llamada. La recepción de parámetros sín los correspondientes argumentos hace que los mismos sean inicializados con NIL. Por ejemplo, la siguiente llamada a función salta dos argumentos, dentro de la lista. Como Ud. puede ver, en los parámetros de salida no es recibido ningún valor o referencia, dando como argumento un valor NIL:

```
? TestProc( "Hola",,,"Ellos" )

FUNCTION TestProc( param1, param2, param3, param4, param5 )
    ? param1, param2, param3, param4
    // Result: Hola NIL NIL Ellos
    RETURN NIL
```

Cuando una rutina es llamada, la función PCOUNT() es actualizada con el valor de la posición del último argumento especificado dentro de la lista. Ello incluye los argumentos saltados, comenzando desde la izquierda hacia el final de la lista. En el ejemplo anterior PCOUNT() retorna 4.

2.4.- Pasando por valor.

Pasando argumentos por valor significa que el mismo es evaluado y este valor es copiado en el parámetro que recibe. Los cambios producidos en el parámetro son locales a la rutina llamada y es perdido cuando esta termina. En Clipper todas las variables, expresiones y elementos de arreglos son pasados por valor si la sintáxis de llamada a función es la implícita. Ello incluye variables que contienen referencias, arreglos, objetos y bloques de código.

Como ejemplo, el siguiente pase de un campo de un fichero base de datos y una variable local hacia un procedimiento por valor:

```
LOCAL nNumber := 10
USE Customer NEW

SayIT( Customer -> Name, nNumber )
? nNumber          // Resultado: 10
RETURN

PROCEDURE SayIT( FiedValue, nValue )
    ? FiedName, ++ nValue        // Resultado: Smith 11
    RETURN
```

La importancia del pase por valor es que la rutina llamada no puede cambiar el dato del llamador al ser cambiado por el valor del parámetro de recepción. Esto incrementa la modularidad por relegar la responsabilidad a la rutina que
llama para determinar si este dato va a ser cambiado.

2.5.- Pasando por referencia.

Pasando un argumento por referencia significa que una referencia hacia el valor del argumento es pasada en vez de una copia del valor. El parámetro receptor entonces apunta a la misma localización en la memoria que el argumento. Si la rutina llamada cambia el valor del parámetro receptor, este también cambia el argumento pasado desde la rutina que llama.

Las variables, así como las variables campo, pueden ser pasadas por referencia si son precedidas por el operador pass-by- reference (paso por referencia) (@) y la función o procedimiento es llamada utilizando la sintáxis de llamada a función. Por ejemplo:

```
nx := ny := 1
Increment( @nx, ny )
? nx              // Resultado: 2
```

```
PROCEDURE Increment( nNumber, nIncrement )
   nNumber := nNumber + nIncrement
   RETURN
```

En este ejemplo, el cambio hecho al parámetro nNumber en la rutina llamada es reflejado en el argumento nx después que el procedimiento Increment() es llamado.

Como Ud. puede ver el pase de argumentos por referencias puede ser muy peligroso si los parámetros son inadvertidamente cambiados por nuevos valores en la rutina llamada. Por eso el pase de parámetros por referencia deberá solo ser usado en casos especiales tales como cuando se requiere retornar más de un valor desde un procedimiento o función usuario. Un buen ejemplo de esto es la función FREAD() la cual pasa una variable buffer por eferencia, llenando la variable con caracteres desde un fichero binario y retornando el número de bytes leídos.

Note que los argumentos pasados a rutinas con la sentencia DO ... WITH son pasados por referencia como implícito. Note también que en otros dialectos el pase de argumentos por referencia es implícito. En Clipper esta práctica es significativamente no recomendada y por tanto el uso de la sentencia DO ... WITH no se recomienda. Todas las invocaciones DO podrán ser reemplazadas por la sintáxis de llamada a una función y las variables pasadas por referencia precedidas por el operador @.

2.6.- Pasando arreglos y objetos como argumento.

Cuando usamos la sintáxis de llamada a una función, las variables que contienen referencias a arreglos y objetos son pasadas por valor como si la variable contuviera valores de cualquier otro tipo de datos. Esto puede parecer confuso ya que puede hacer parecer que el paso de parámetros por referencia implica un direccionado indirecto. Pero cuando una variable conteniendo una referencia es pasada hacia una rutina, una copia de la referencia es pasada en vez de la referencia misma.

Si por ejemplo, un arreglo es pasado a una rutina por valor y la rutina llamada cambia el valor de uno de los elementos del arreglo,el cambio es reflejado en el arreglo después del retorno, ya que el cambio fue hecho hacia el arreglo actual por vía de la referencia hacia éste que fue copiada. Si, por otra parte, una referencia hacia un nuevo arreglo es asignada a una variable pasada por valor, dicha referencia es descartada por el llamador en el retorno y la referencia al arreglo contenida en la variable original no es afectada.

Recíprocamente, si una variable conteniendo una referencia hacia un arreglo u objeto es pasada por referencia precedida la variable argumento con el operador (@), cualquier cambio a la referencia es reflejado en la rutina que llamó después del retorno. Por ejemplo:

```
// Cambia el valor en un elemento de arreglo
a := { 1, 2, 3 }
ChangeElement( a )
? a[1]          // Resultado: 10

// Cambia el arreglo; no trabaja
a := { 1, 2, 3 }
ChangeArray( a )
? a[1]          // Resultado: 1

// Cambia el arreglo por referencia
a := { 1, 2, 3 }
ChangeArray( @a )
? a[1]          // Resultado: 4

FUNCTION ChangeElement( aArray )
   aArray[1] := 10
   RETURN NIL
```

```
FUNCTION ChangeArray( aArray )
   aArray := { 4, 5, 6 }
   RETURN NIL
```

Aunque esta conducta aparece diferente en versiones anteriores, este ya no es el caso. La confusión en la diferencia de tipos de arreglos son resueltas en Clipper 5.0. En versiones anteriores de Clipper usted no podia asignar, retornar o transportar una referencia cualquiera hacia un arreglo excepto pasando este hacia un procedimiento o función usuario, es por ello que nunca existió la vía de conocer cómo estaban en la actualidad los arreglos pasados a subrutinas. Ellos siempre se comportaron como si fueran pasados por referencia y así fue entendido. En Clipper 5.0 las referencias a arreglos son tratadas al igual que otro tipo de datos, de este modo revelando las actuales y verdaderas reglas del pasado de parámetros.

2.7. - Chequeando argumentos.

En Clipper no existe chequeo de argumentos y por lo tanto el número de parámemetros no tiene que ser igual al número de argumentos pasados. Los argumentos pueden ser saltados o no completar la lista de parámetros. Un parámetro que no recibe valor o referencia es inicializado con NIL. Si los argumentos son especificados, PCOUNT() retorna la posición del último argumento pasado.

Ya que los argumentos pueden ser saltados, PCOUNT() no siempre da la medida exacta de que argumentos han sido especificados. Para asegurarse de ello, se puede comparar un parámetro con NIL y si son iguales, se puede suplir por un valor implícito o generar un error de argumento. El siguiente ejemplo demuestra este concepto:

```
FUNCTION TestParam( param1, param2, param3 )
  IF Param2 = NIL
     ? "Parametro no fue pasado"
     param2 := "valor por defecto"
  ENDIF
  .
  . <statements>
  .
  RETURN NIL
```

Además de comprobar el NIL, VALTYPE() puede ser usada para comprobar un valor recibido por un parámetro así como el tipo de dato, ejemplo de esto:

```
FUNCTION TestParam( cParam1, nParam2 )
  IF VALTYPE( nParam2 ) != "N"
     ? "El parametro no fue pasado o el tipo es inválido"
  ENDIF
  .
  . <statements>
  .
  RETURN NIL
```

Nota: Recuerde que VALTYPE() se usa en lugar de TYPE() para comprobar el tipo de dato de los parámetros declarados. Un TYPE() aplicado a un parámetro no declarado, retorna siempre "U".

2.8.- Pasando argumentos desde la línea de comando del DOS.

Los argumentos pueden ser pasados directamente desde la línea de comandos del DOS a los procedimientos principales de sus programas. Cada uno de los parámetros son especificados con una sentencia PARAMETERS o

declarados como parte de una declaración de un procedimiento principal. Recuerde, si el procedimiento principal esta declarado en el fichero programa principal (.prg) debe ser compilado con la opción /N.

Los argumentos pasados son todos recibidos como una cadena de caracteres. Múltiples argumentos son especificados separados cada uno por espacio. Si un argumento contiene espacio, este debe ser encerrado entre comillas para ser pasado como una cadena. Por ejemplo, la siguiente línea de comando del DOS, pasa dos argumentos cuando es invocado PROG.EXE:

C>PROG "Clipper Compiler" 5

Los parámetros recibidos por el procedimiento principal desde la línea de comandos del DOS, podrán ser comprobados con PCOUNT() para confirmar el número exacto de parámetros que han sido pasados.

2.9.- Retornando valores desde las funciones usuario.

Por definición, una función usuario debe retornar un valor. Para hacer esto se debe especificar un argumento en la sentencia RETURN. Una sentencia RETURN ejecuta dos acciones: primero, esta termina el procesamiento de la rutina actual transfiriendo el control hacia la rutina que la llamó; y segundo, si la rutina actual es una función usuario esta retorna cualquier valor especificado como argumento de la sentencia RETURN hacia la rutina que la llamó. El valor retornado puede ser una constante o una expresión. Por ejemplo:

```
RETURN ("Hoy es " + DTOC(DATE())
```

En Clipper 5.0, una función usuario puede retornar un valor de cualquier tipo de dato incluyendo arreglos, objetos, bloques de códigos y NIL. Por ejemplo, la siguiente función usuario retorna un arreglo a la rutina que la llamó:

```
FUNCTION NumArrayNew
  LOCAL aNumArray := AFILL( ARRAY(10), 1, 1 )
  RETURN aNumArray
```

La sentencia RETURN puede aparecer en cualquier parte de la definición de la función usuario permitiendo que el proceso sea terminado antes del final de la función. Por ejemplo, el siguiente fragmento de código termina el procesamiento si una condición es verdadera (.T.) y continua si la condición es falsa (.F.):

```
FUNCTION OpenFile( cDBF )
  USE ( cDBF )
  IF NETERR()
    RETURN (.F.)
  ELSE
  .
  . <statements>
  .
  RETURN (.T.)
```

Note que RETURN es limitado por el hecho que este puede retornar solo un valor. Más de un valor puede ser retornado si los argumentos son pasados por referencia, aunque esto no es una solución adecuada. Una estructura de datos agregada puede ser definida como un arreglo conteniendo otros arreglos, y los arreglos, como los mencionados anteriormente, pueden ser pasado a lo largo de un programa como un simple valor.

Si una función usuario retorna un valor que no es usado, este es descartado. Por esta razón, se puede especificar una función usuario como una sentencia.

2.10.- Recursión.

Un procedimiento o función usuario es recursivo si este contiene una llamada a sí mismo. Esto puede ser de forma directa o indirecta cuando una función llama a otra función que repite la llamada a la función original.

Por ejemplo, el siguiente ejemplo es una función usuario que usa la recursión para calcular el factorial de un número. factorial de 3, por tanto, es 1 * 2 * 3 el cual es 6:

```
FUNCTION Factorial( nFactorial )
  IF nFactorial = 0
    RETURN (1)
  ELSE
    RETURN ( nFactorial * Factorial( nFactorial - 1 ))
  END
```

3.- Estructuras de Control.

Clipper soporta varias estructuras de control que cambian el flujo secuencial de la ejecución de una programa. Estas estructuras permiten ejecutar un código a partir de unas condiciones lógicas y repetidamente ejecuta el código cualquier número de veces.

En Clipper, todas las estructuras de control pueden estar anidadas dentro de otra estructura de control, tanto como este anidamiento sea apropiado. Las estructuras de control tienen una estructura de inicio y de fin, y una estructura anidada debe estar entre las sentencias de inicio y de fin de la estructura en la cual ella está anidada. Esta sección resume toda la estructura de control del lenguaje Clipper, dando ejemplos y sugiriendo como usar cada una de ellas.

3.1.- Estructuras Cíclicas.

Las estructuras cíclicas son designadas para cuando se quiera ejecutar una sección de código más de una vez. Por ejemplo, se quiere imprimir un reporte cuatro veces. Claro, Ud. podrá simplemente repetir cuatro veces el comando REPORT FORM, pero esto puede ser un problema y no podrá resolverse si el número de reportes fuera variable.

3.1.1.- FOR ... NEXT.

La estructura de control FOR ... NEXT, o FOR cíclico, repite una sección de código un número particular de veces. Para resolver el problema anterior, se podría hacer esto:

```
FOR i := 1 TO 4
  REPORT FORM Accounts TO PRINTER
NEXT
```

FOR ... NEXT es una estructura de control típica de Clipper. Esta consiste de una sentencia que define la condición de la estructura y marca su inicio. Esta sentencia es seguida por una o más sentencias ejecutables, que representan el cuerpo de una estructura y, finalmente, una sentencia de fin de la estructura.

FOR ... NEXT comienza un ciclo con la inicialización de una variable contador con un valor numérico. El contador es incrementado cada vez que continue el ciclo, después el cuerpo de la sentencia es ejecutado hasta que éste alcance el valor especificado. Por defecto, como en el ejemplo anterior, el incremento del valor es 1, pero el incremento puede también ser especificado como parte de la sentencia FOR. Además, todos los parámetros del FOR pueden ser expresiones numéricas seguidas por un número variable de iteraciones. FOR ... NEXT es comúnmente usado para procesar arreglos elemento a elemento.

3.1.2.- DO WHILE ... ENDDO.

Otro tipo de ciclo es basado en una condición más que en número particular de repeticiones. Por ejemplo, suponga que Ud. busca ejecutar un proceso complejo usando artículos en un fichero de base de datos para un particular número de clientes. Sin una estructura de ciclo condicional, esto podría ser muy difícil, si no imposible, para resolver este problema.

La estructura de control DO WHILE ... ENDDO (también llamada ciclo DO WHILE), procesa una sección de código, mientras una condición especificada sea verdadera (.T.). Para resolver el problema de la base de datos, se podrá hacer lo siguiente:

```
USE Accounts INDEX Accounts
SEEK 3456
DO WHILE Accounts->AccNum = 3456
  .
  . <procesing statements>
  .
  SKIP
ENDDO
```

Este ejemplo ilustra un uso típico del DO WHILE para procesar los artículos de un fichero de base de datos. Note que el SKIP es usado como parte del cuerpo para avanzar el puntero al próximo artículo. En un ciclo DO WHILE, el cuerpo del ciclo debe contener alguna sentencia que altere la condición del ciclo; de lo contrario, el ciclo se va a ejecutar siempre.

3.1.3.- EXIT y LOOP.

EXIT y LOOP son sentencias especiales que solo pueden ser usadas dentro de un ciclo DO WHILE o FOR. EXIT tranfiere el control fuera del ciclo y LOOP transfiere el control al inicio del ciclo. Estas sentencias son usadas como parte de una estructura de control condicional para controlar el funcionamiento bajo circunstancias inusuales.

3.2.- Estructuras de toma de decisión.

Las estructuras de toma de decisiones permiten ejecutar una o más sentencias de programa basada en una condición. Por ejemplo, Ud. puede desear ejecutar diferentes funciones a partir de un menu de opciones. Clipper tiene dos estructuras semejantes, IF ... ENDIF y DO CASE ... ENDCASE, pero ellas son idénticas en su funcionamiento.

3.2.1.- IF ... ENDIF.

La estructura de control IF ... ENDIF ejecuta una sección de código si la condición especificada es verdadera (.T.). La estructura puede también especificar un código alternativo para ejecutar si la condición es falsa (.F.).

El siguiente ejemplo ilustra el IF ... ENDIF en una función usuario que comprueba si un arreglo esta vacío.

```
FUNCTION ZeroArray( aEntity )
  IF VALTYPE( aEntity ) = "A" .AND. EMPTY( aEntity )
    RETURN (.T.)
  ENDIF
  RETURN (.F.)
```

El próximo ejemplo usa el IF ... ENDIF para procesar un menu de opciones:

```
IF nChoice = 1
  Func1()
ELSEIF nChoice = 2
```

```
  Func2()
ELSEIF nChoice = 3
  Func3()
ELSE
  QUIT
ENDIF
```

ELSEIF especifica una condición alternativa a comprobar, si la condición anterior no se cumplió seguida de múltiples niveles de control. ELSE es un camino alternativo a ejecutar si ninguna de las condiciones anteriores en la estructura fueron cumplidas.

3.2.2.- DO CASE ... ENDCASE.

DO CASE e IF son estructuras equivalentes con una ligera diferencia en la representación de su sintáxis. Por ejemplo, el siguiente segmento de DO CASE es equivalente funcionalmente al anterior ejemplo del IF:

```
DO CASE
  CASE nChoice = 1
    Func1()
  CASE nChoice = 2
    Func2()
  CASE nChoice = 3
    Func3()
  OTHERWISE
    QUIT
ENDCASE
```

Ninguna estructura de toma de decisión tiene ventajas sobre la otra. Por lo cual el uso de una forma puramente es preferencia personal.

3.3.- Estructuras manipuladoras de Errores.

BEGIN SEQUENCE ... END es una estructura de control especializada regularmente usada para manipulación de secciones y errores de ejecución. Refiérase al capítulo de Manipulación de Errores de Ejecución en el libro "Programming and Utilities" para más información acerca de esta estructura de control.

4.- Variables.

En esta sección se decribe la naturaleza básica de las variables en Clipper. Las variables son portadoras de valores que tienen definidos un tiempo de vida y una visibilidad, y tienen un nombre. Cuando un nombre de variable es refe-
renciado, el valor de esta es retornado.

En Clipper 5.0 existen varios tipos de variables. Las variables son organizadas dentro de "storage classes" que determina cómo la variable es almacenada, el tiempo de vida de ésta, y en que lugar de un programa este nombre puede ser referenciado. Cada "storage classes" tiene una sentencia que declara el nombre de la variable o crea la variable en ejecución.

4.1.- Alcance literal Vs. Alcance dinámico de variables.

Si Ud. está familiarizado con el Clipper, conoce de versiones anteriores las variables soportadas PUBLIC, PRIVATE y FIELD. Estas variables tienen que ser referenciadas con un alcance dinámico. Las variables de alcance dinámico,

incluyendo sus nombres son creadas y mantenidas completamente en ejecución. Nada es resuelto en tiempo de compilación y existe herencia automática de variables en llamadas a procedimientos y funciones usuario. Estas clases de variables son característica del sistema de intérpretes y tienen existencia en Clipper para ser compatibles con otros dialectos.

En Clipper 5.0, dos nuevas clases de variables han sido introducidas que son denominadas de alcance literal. Estas clases de variables son resueltas completamente en tiempo de compilación y tienen reglas de alcance rígidamente definidas, como se describe a continuación.

Las variables de alcance literal han sido introducidas por diversas razones:

- Las variables de alcance dinámico violan el principio de la modularidad ya que para entender la operación de una rutina, Ud. debe entender la operación de todas las rutinas que llaman a ésta. Cada rutina es una construcción modular de programa que podrá ser entendida en el contexto de esta propia definición, y localizada dentro de un fichero programa (.prg).

- Las variables con alcance literal son más eficientes y mucho más duraderas que las variables de alcance dinámico ya que son resueltas completamente en tiempo de compilación. Las variables de alcance dinámico, por el contrario, deben ser resueltas cada vez que son referenciadas en el programa.

Las variables con alcance literal son altamente recomendadas y se podrá eventualmente reemplazar todos los usos de variables de alcance dinámico.

4.2.- Tiempo de vida y visibilidad de una variable.

Durante la ejecución, una variable, si bien es declarada en tiempo de compilación o referenciada dentro de una sentencia ejecutable, no existe realmente hasta que en alguna porción de memoria interna de la computadora, se localice su valor. Esto se conoce como creación de la variable (instantiating). Algunas variables son creadas automáticamente, mientras que otras deben ser creadas explícitamente.

Una vez creada, una variable continúa existiendo (y posee un valor, si este fue asignado), hasta que esta memoria sea liberada. Algunas variables son liberadas automáticamente, mientras que otras deben ser explícitamente liberadas. Algunas variables nunca podrán ser liberadas. La duración de la vida de las variables es conocido como su tiempo de vida.

Se conoce como visibilidad a las condiciones bajo las cuales una variable es accesible a un programa durante la ejecución. Algunas variables, aunque hayan sido creadas y asignado un valor, no pueden ser visibles bajo ciertas condiciones.

Durante la ejecución, es posible que un nombre simple de variable este asociado simultáneamente con diferentes variables, algunas de las cuales o todas pueden ser visibles. Las declaraciones pueden ser usadas para asegurar que la ocurrencia de un nombre particular en su código fuente refiera a una variable deseada.

4.3.- Declarando variables.

Las declaraciones de variables no son sentencias ejecutables. En cambio, ellas declaran en tiempo de compilación, los nombres de variables de programas y el informe de asunciones del compilador, que puede ser hecho de ellas. Aunque las declaraciones no son ellas mismas ejecutables, las asunciones de ellas producen una afección del código objeto generado por el compilador para las referencias a las variables declaradas.

Las variables pueden ser declaradas nombrándose en sentencias de declaración STATIC, LOCAL, MEMVAR y FIELD. Ellas pueden además nombrarse al ser declaradas como parámetros (esto es una lista de ellas) en sentencias PROCEDURE o FUNCTION o en una definición de bloques de código.

Las declaraciones son opcionales para algunos tipos de variables (privadas, públicas y de campos) y obligatorias para otras (estáticas, locales y parámetros declarados).

Durante la ejecución, es posible que para un simple nombre de variable estén asociados simultáneamente diferentes variables. Las declaraciones pueden ser usadas para asegurar la aparición de un nombre particular en el código fuente referido a la variable deseada en tiempo de ejecución.

En conjunción con la opción /W del compilador, las declaraciones en tiempo de compilación siempre se chequean para las variables no declaradas. Todas las sentencias de declaración deben ocurrir antes de las sentencias ejecutables.

4.4.- El alcance de una declaración.

El alcance de una declaración es quien determina la parte del programa hacia la cual la declaración es aplicada. El alcance de la declaración es determinado literalmente. Es decir, las declaraciones se utilizan solo en la compilación de ciertas secciones del código fuente. (Note que el alcance de una declaración no es necesariamente igual al de visibilidad o tiempo de vida de esas variables declaradas).

Para determinar el alcance de una declaración, el código fuente es visto como una serie de unidades literales discretas: ficheros, procedimientos y bloques de código. Un fichero es todo el código fuente dentro de un simple archivo en disco sometido a compilación. Note que el código incluído por vía de la directiva al pre-procesador #INCLUDE es considerado parte del mismo archivo. Un procedimiento es todo el código fuente desde una sentencia PROCEDURE o FUNCTION hasta la próxima sentencia PROCEDURE o FUNCTION o hasta el fin de fichero. Un bloque es todo el código fuente dentro de los delimitadores de bloque que lo definen. Algunas unidades pueden ser anidadas dentro de otras; por ejemplo, un procedimiento puede ocurrir dentro de un fichero, y un bloque
de código puede ocurrir dentro de un procedimiento o dentro de otro bloque de código.

Una declaración tiene alcance para la unidad lexical en la que ésta ocurre y alguna unidad anidada. Por ejemplo, una declaración que ocurre dentro de una definición de procedimiento se aplica a éste y a cualquier bloque dentro del mismo. Una declaración que ocurre dentro de un bloque de código (un parámetro bloque) se aplica a este bloque de código y a cualquiera anidado dentro de él.

Las declaraciones STATIC, FIELD y MEMVAR pueden aparecer además fuera de cualquier procedimiento o bloque de códigos, especificándose antes de cualquier sentencia PROCEDURE o FUNCTION en el fichero fuente, en cuyo caso la declaración se aplica a todo el fichero fuente. Las variables declaradas de este modo son "file-wide scope". Advierta que el fichero programa (.prg) contentivo de las variables declaradas como " file-wide scope", debe ser compilado con la opción /N.

Una declaración en una unidad interna puede declarar una variable con el mismo nombre que una declarada en una unidad externa. En este caso, la declaración interna reemplaza la externa, pero solo dentro de la unidad interna. Por ejemplo, una declaración aplicada a un fichero entero puede ser reemplazada por
una declaración en uno de los procedimientos dentro del mismo. La declaración reemplazada afecta solo al procedimiento en el cual esta ocurrió.

4.5.- Referenciando a variables.

En el código fuente del programa, las variables son referenciadas por su nombre. Cuando el compilador ve una ocurrencia de un nombre particular, genera un código objeto para asignar o actualizar el valor de la variable de ese nombre.

4.6.- Referencias ambiguas a variables.

Durante la ejecución, es posible que para un simple nombre estén asociados simultáneamente diferentes nombres de variables, algunas de las cuales, o todas, pueden ser visibles. En ausencia de una declaración o un alias implemen

tado explícitamente, es imposible para el compilador decir cual nombre de variable en particular puede referirse en tiempo de ejecución del programa. La ocurrencia del mismo nombre es conocida como una "ambiguous variable
reference" (referencias ambiguas a variables).

Cuando el compilador encuentra una referencia ambigua a una variable, genera un código objeto especial (lookup code) el cual cuando es ejecutado chequea si una variable existe con igual nombre. Si el nombre existe, este chequea que tipo de variable es y si ésta es visible. El orden en que las variadas posibilidades son chequeadas depende de cómo la variable fue referenciada.

4.6.1.- Evitando referencias ambiguas a variables.

El evitar una referencia ambigua en tiempo de compilación, hace que el compilador siempre genere más eficiente el código objeto. Esto además, mejora la independencia del programa ya que las referencias ambiguas se comportan diferentemente en tiempo de ejecución dependiendo de una variedad de circunstancias.

Las declaraciones pueden ser usadas para asegurar que la ocurrencia de un nombre particular en el código fuente se refiera a la variable deseada. Alternativamente, las referencias a variables pueden ser explícitamente implementadas precediendo las mismas con el operador alias.

La opción /V del compilador causa que las referencias ambiguas a variables sean tratadas como si estas fueran precedidas por el implementador especial MEMVAR->. La opción /W del compilador puede ser usada para generar advertencias sobre las referencias ambiguas a variables.

4.6.2.- Implementando referencias a variables.

El operador alias (->) puede ser usado explícitamente para implementar una referencia a variable. Las siguientes combinaciones son legales:

<alias>-><name>

Referencia a un campo en el area de trabajo designada por <alias>. Un error de ejecución ocurre si <name> no es un campo en el area de trabajo designada.

FIELD-><name>

Referencia a un campo en el area de trabajo actualmente seleccionada. Un error de ejecución ocurre si <name> no es un campo en el area de trabajo actualmente seleccionada.

MEMVAR-><name>

Referencia a cualquier variable PRIVATE (si existe) o de lo contrario a una variable PUBLIC. Si la variable no existe y la referencia fue un intento de asignar valor, una variable PRIVATE es creada en el nivel actual de activación. En otro caso ocurre un error de ejecución.

4.7.- Creación e inicialización.

Los valores iniciales pueden ser suministrados en sentencias en las cuales se especifica la creación de variables usando inicializadores. Las siguientes sentencias permiten la inicialización:

- STATIC
- LOCAL

- PRIVATE
- PUBLIC

Las sentencias FIELD y MEMVAR no permiten inicializar ya que ellas no especifican la creación de variables.

Los inicializadores para variables estáticas deben ser expresiones constantes en tiempo de compilación. Esto es, las expresiones deben consistir enteramente de constantes y operadores simples (no variables o llamadas a funciones).

Los inicializadores para otras variables pueden ser cualquier expresión válida Clipper.

Los inicializadores para variables estáticas son computados en tiempo de compilación y son asignadas a las variables antes del comienzo de la ejecución.

Los inicializadores para otras variables son computados en el momento de creación de la variable y asignados a estas inmediatamente después. Note que estos permiten que una variable privada pueda ser inicializada con el valor que tuvo una privada ya existente o variable pública con el mismo nombre.

Si el inicializador no es especificado explícitamente, las variables asumen el valor NIL, excepto para variables públicas que asumen .F. Por ejemplo:

```
STATIC lFirstTime := .T.

LOCAL i := 0, j:= 1, nMax := MAXROW()

PRIVATE cString := "This is a string"
```

4.8.- Variables locales.

Las variables locales deben ser declaradas explícitamente en tiempo de compilación con la sentencia LOCAL. Las variables locales pueden ser declaradas como sigue:

```
FUNCTION SomeFunc
  LOCAL  nVar := 10, aArray[10][10]
  .
  . <executables statements>
  .
  NextFunc()
  RETURN (.T.)
```

En el ejemplo anterior, la variable nVar ha sido declarada como LOCAL y definida con un valor de 10. Cuando la función NextFunc() es ejecutada, nVar todavía existe pero no se puede accesar. Tan pronto como la ejecución de SomeFunc() es completada, nVar es destruída. Cualquier variable con el mismo nombre en el programa que llama no es afectada.

Las variables locales son creadas automáticamente cada vez que es es activado el procedimiento en el que fueron declaradas. Ellas continúan existiendo y mantienen sus valores hasta el fin de la activación (es decir, hasta que el procedimiento retorna el control al código que lo invocó). Si un procedimiento es invocado recursivamente (llamadas a si mismo), cada activación del procedimiento crea un juego de variables locales.

La visibilidad de una variable local es idéntica al alcance de su declaración. Esto es, la variable es visible en cualquier parte del código fuente en el que fue declarada. Si un procedimiento es invocado recursivamente, solo las locales creadas para la más reciente activación, son visibles.

4.9.- Variables estáticas.

Las variables estáticas trabajan muy semejante a las locales, pero retienen sus valores en toda la ejecución y pueden ser declaradas con un inicializador. Las variables estáticas deben ser declaradas explícitamente en tiempo de compilación con la sentencia STATIC.

Su alcance depende de donde estas sean declaradas. Si ellas son declaradas dentro del cuerpo de una función, su alcance es limitado a esta función. Si ellas son declaradas fuera de una función, su alcance es para toda la compilación. Compile con /N para suprimir la declaración implícita de procedimiento. Por ejemplo:

```
FUNCTION SomeFunc
  STATIC myVar := 10
  .
  . <statements>
  .
  NextFunc()
  RETURN (.T.)
```

En este ejemplo, la variable myVar ha sido declarada como estática e inicializada con un valor de 10. Cuando la función NextFunc() es ejecutada, aún existe pero no puede ser accesada. De manera diferente a las variables declaradas como LOCAL o PRIVATE, myVar puede todavía existir cuando SomeFunc() complete su ejecución; como quiera que sea, ella no puede ser accesada excepto por las subsiguientes ejecuciones de SomeFunc().

Las variables estáticas son creadas automáticamente antes de comenzar la ejecución. Ellas continúan su existencia y retienen sus valores durante toda la ejecución del programa. Ellas no pueden ser borradas.

Un valor inicial puede ser suministrado en una sentencia STATIC. Si el valor inicial no es suministrado la variable estática es inicializada con NIL.

4.10.- Variables privadas.

Para este tipo de variable la declaración es opcional. Si es deseado ellas pueden ser declaradas explícitamente en tiempo de compilación con la sentencia MEMVAR.

Las variables privadas son creadas dinámicamente en tiempo de ejecución con las setencias PRIVATE y PARAMETERS. Adicionalmente asignar un valor a una variable no creada anteriormente creará automáticamente una variable privada. Una vez creada, una variable privada continúa su existencia y retiene sus valores hasta concluir el procedimiento en la cual fue creada (es decir, hasta que el procedimiento que la creó retorne al que la invocó). En este momento ella es automáticamente borrada. Las variables privadas pueden ser además borradas explícitamente usando la sentencia RELEASE.

Es posible la creación de una nueva variable privada con igual nombre que una existente anteriormente. Como quiera que sea, la nueva (duplicada) variable puede solo ser creada en un nivel de activación más bajo que la ya existente. La nueva privada oculta cualquier otra existente (o pública, ver abajo) con el mismo nombre.

Una vez creada, una variable privada es visible en todo el programa hasta que sea borrada (explícita o automáticamente, ver explicación anterior) u oculta por la creación de cualquier variable privada con el mismo nombre. En este caso, la nueva (duplicada) privada oculta cualquier existencia privada con el mismo nombre y la privada existente llega a ser inaccesible hasta que la nueva privada es borrada.

En otros términos, una variable privada es visible dentro del procedimiento que la creó y cualquier procedimiento llamado por éste, a menos que una llamada a procedimiento cree en él mismo una variable privada con el mismo nombre.Por ejemplo:

```
FUNCTION SomeFunc
  PRIVATE myVar := 10
  .
```

```
  . <statements>
  .

  NextFunc()
  RETURN (.T.)
```

En este ejemplo, la variable myVar ha sido creada como una variable privada e inicializada con el valor 10. Cuando la función NextFunc() es ejecutada, myVar aún existe y, a diferencia de una variable local, puede ser accesada por NextFunc(). Cuando SomeFunc() termina, myVar es borrada y cualquier definición previa se hace accesible de nuevo.

Cuando una variable privada es creada y no es inicializada con un valor, asumirá el valor NIL.

4.11.- Variables públicas.

Las declaraciones para variables públicas son opcionales. Si es deseado, ellas pueden ser declaradas explícitamente en tiempo de compilación con la sentencia MEMVAR.

Las variables públicas son creadas dinámicamente en tiempo de ejecución con la sentencia PUBLIC. Ellas continúan su existencia y retienen sus valores hasta el fin de la ejecución, a menos que ellas sean explícitamente borradas con la
sentencia RELEASE.

Es posible la creación de una variable privada con el mismo nombre que una variable pública existente. No es posible, de cualquier modo, crear una variable pública con el mismo nombre de una privada ya existente.

Una vez creada una variable pública es visible en todo el programa hasta que ésta es explícitamente borrada u oculta por la creación de una variable privada con el mismo nombre. En este caso, la nueva variable privada oculta la existencia de la variable pública del mismo nombre y ésta por tanto llega a hacerse inaccesible hasta que la nueva privada es borrada. Por ejemplo:

```
FUNCTION SomeFunc
  PUBLIC myVar := 10
  .
  . <statements>
  .
  NextFunc()
  RETURN (.T.)
```

En este ejemplo, myVar ha sido creada como una variable pública e inicializada con valor 10. Cuando la función NextFunc() es ejecutada, myVar aún existe y puede ser accesada. A diferencia de las variables LOCAL y PRIVATE, myVar aún existe cuando la ejecución de SomeFunc() es completada.

La creación de una variable pública cuando la variable aún no existe, crea una variable nueva lógica con un valor inicial de .F.

4.12.- Variables campos.

La declaración de variables campos es opcional. Si es deseado, ellas pueden ser declaradas explícitamente en tiempo de compilación con la sentencia FIELD. Las variables FIELD son realmente sinónimos de los campos de las bases de datos de igual nombre. Por consiguiente, ellos típicamente existen y poseen valores antes que el programa sea ejecutado, y continúan existiendo después que el programa termina. El valor de una variable campo depende, en cualquier momento, del artículo actual de la base de datos asociada. Una vez que el fichero de base de datos es abierto, las correspondientes variables campos son visibles en todo el programa, ellas se mantienen visibles hasta que el ficherode bases de datos es cerrado.

Las variables FIELD son inicializadas vacias (empty) cuando los artículos son adicionados en el fichero de base de datos.

Cuando una variable es declarada como FIELD, las subsiguientes referencias no implementadas a esta variable causarán que el programa mire para un campo de base de datos en el área de trabajo actual con ese nombre.

5.- Expresiones.

En Clipper, todos los elementos de datos (items) son identificados por un tipo, con el propósito de formar expresiones. Los items de datos básicos (es decir, variables, constantes y funciones) son asignados a tipos de datos dependiendo de cómo el item es creado. Por ejemplo, el tipo de dato de una variable FIELD es determinado por la estructura de su base de datos, el tipo de datos de un valor constante depende de cómo éste es formado y el tipo de datos de una función es definido por el valor que retorna.

Estos items básicos de datos son simples expresiones en lenguaje Clipper. Expresiones más complejas son formadas por combinación de items básicos con operadores. Esta sección define todos los tipos de datos válidos para Clipper y los operadores que usted puede usar para formar expresiones, así como algunas reglas especiales que usted necesita conocer.

5.1.- Tipos de datos

Los tipos de datos soportados en el lenguaje Clipper son:

- Arreglo
- Caracter
- Bloque de códigos
- Númerico
- De fecha
- Lógico
- Memo
- NIL

En Clipper una estructura de arreglo es un tipo de dato particular, por lo que hay distintos tipos de operaciones para arreglos que se invalidan para otros tipos de datos. El bloque de códigos es también un tipo de dato pero las operaciones que usted puede efectuar usando los mismos es limitada. Arreglos y bloques de códigos serán discutidos en secciones separadas en este capítulo.

Esta sección trata cada uno de los tipos de datos simples dotándolo a usted de la siguiente información: cuándo se usa el tipo de datos; cómo formar constantes o literales; que limitaciones hay en efecto; y que operaciones son permisibles.

5.1.1.- Caracter

El tipo de dato de caracter es utilizado para identificar un item de datos que usted quiere manipular como una cadena de longitud fija. El conjunto de caracteres Clipper es definido como el conjunto íntegro de caracteres ASCII imprimibles (es decir, CHR(32) hasta CHR(126)), el conjunto de caracteres ASCII gráficos extendido (es decir,

CHR(128) hasta CHR(255)) y el caracter nulo, CHR(0). Vea en este libro Apéndice de Caracteres ASCII para que complete la tabla de códigos ASCII.

Una cadena de caracteres válida está formada por una combinación de cero o más caracteres en el conjunto de caracteres definido, con un máximo por cadena de 65535 (es decir, 64k menos un caracter usado como terminador nulo).

Las literales de cadena de caracteres, o constantes, están formadas por una cadena de caracteres válida encerradas por un par de delimitadores. En el lenguaje Clipper los pares de delimitadores son los siguientes:

- Dos simples apóstrofes (por ejemplo, 'uno abre otro cierra')
- Un par de comillas (por ejemplo, "uno abre otro cierra")
- Corchetes izquierdo y derecho (por ej., [izquierdo abre y derecho cierra])

Nota Para empezar una cadena nula, use solo el par de delimitadores sin incluir caracteres (incluyendo el espacio). Por ejemplo, "" y [] ambos representan una cadena nula.

Ya que todos los caracteres delimitadores designados forman parte del conjunto de caracteres válido, los mismos caracteres delimitadores pueden ser parte de una cadena. Si usted quiere incluir cualquiera de estos caracteres en una literal de cadena, debe usar un caracter alternativo para el delimitador. Por ejemplo, las siguientes cadenas:

. I don't want to go.

podrá ser expresado como:

. "I don't want to go."

similarmente, la cadena:

. She said, "I have no idea."

puede ser representada con:

. 'She said, "I have no idea."'

Esta restricción no es una limitación ya que usted tiene tres pares de delimitadores para elegir.

La siguiente tabla da una sinópsis de todas las operaciones en el lenguaje Clipper que son válidas para el tipo de dato de caracter. Estas operaciones actúan en una o más expresiones de caracter para producir un resultado. El tipo de dato del resultado no es necesariamente de caracter.

Nota: El tipo de dato MEMO que se discutirá posteriormente en esta sección, es usado para representar un dato de caracter de longitud variable. Este es un verdadero tipo de dato de longitud variable que puede solo existir en la forma de campo de una base de datos. Las operaciones de caracter listadas en esta sección pueden ser además usadas en los campos MEMO.

Tabla 1-1: Operaciones de caracter

Operación	Descripción
+	Concatenación
-	Concatenación sin intervenir espacios
=	Comparar por igualdad

==	Comparar por exacta igualdad
!=, <>, #	Comparar por desigualdad
<	Compara por ASCII menor a
<=	Compara por ASCII menor o igual a
>	Compara por ASCII mayor a
>=	Compara por ASCII mayor o igual a
= ó STORE	Asignación
:=	Asignación en línea
+=	Concatenación en línea (+) y asignación
-=	Concatenación en línea (-) y asignación
$	Chequeo de existencia de subcadena
REPLACE	Reemplaza valor en campo
ALLTRIM()	Elimina espacios a cada lado
ASC()	Convierte un No. ASCII al código equivalente
AT()	Localiza posición de subcadena
CTOD()	Convierte a fecha
DESCEND()	Convierte a forma complementada
EMPTY()	Chequeo de nulo o blanco
ISALPHA()	Chequeo de la letra inicial
ISDIGIT()	Chequeo para el dígito inicial
ISLOWER()	Chequeo para la letra inicial minúscula
ISUPPER()	Chequeo para la letra inicial mayúscula
LEFT()	Extrae subcadena desde la izquierda
LEN()	Calcula longitud
LOWER()	Convierte letras a mayúscula
LTRIM()	Borra espacios delante
PADC()	Rellena con espacios delante y detrás
PADL()	Rellena con espacios delante
PADR()	Rellena con espacios detrás
RAT()	Localiza posición de subcadena desde derecha
REPLICATE()	Duplica
RIGHT()	Extrae subcadena desde la derecha
RTRIM()	Borra espacios detrás
SOUNDEX()	Convierte a soundex equivalente
SPACE()	Crea una cadena en blanco
STRTRAN()	Busca y reemplaza subcadenas
STUFF()	Reemplaza subcadenas
SUBSTR()	Extrae subcadenas
TRANSFORM()	Convierte a cadenas formateadas
TYPE()	Evalúa tipo de datos usando macrosustitución
UPPER()	Convierte letras a mayúscula
VAL()	Convierte a numérico
VALTYPE()	Evalúa directamente el tipo de dato

Las cadenas de caracteres son comparadas de acuerdo al ASCII compulsando en secuencia. El campo MEMO puede además ser comparado como cadena de caracteres. Cuando EXACT es OFF dos cadenas de caracteres son comparadas de acuerdo a la siguiente regla; asuma dos cadenas de caracteres A y B donde la expresión a chequear es (A=B):

- Si B es nula retorna verdadero (.T.)
- Si LEN(B) es mayor que LEN(A) retorna falso (.F.)
- Compara todos los caracteres en B con A. Si todos los caracteres en B son iguales a A, retorna verdadero (.T.); de lo contrario retorna falso (.F.)

Con EXACT en ON, dos cadenas deben igualarse exactamente, excepto para blancos al final.

5.1.2.- MEMO

El tipo de dato MEMO es utilizado para representar un dato de caracter de longitud variable. Este es un verdadero tipo de dato de longitud variable que solo existe en la forma de campo de una base de datos. Un campo MEMO contiene 10 caracteres en el fichero base de datos el cual es usado como puntero al actual dato que es almacenado en un fichero separado (.dbt).

A pesar del hecho de que la longitud pueda variar en un campo MEMO entre un artículo y otro, los campos MEMO son idénticos a los campos de caracter. El juego de caracteres es idéntico, la limitación impuesta es de un máximo de 65535 caracteres y las comparaciones son ejecutadas del mismo modo.

Debido a que el tipo MEMO es aplicado solo a campo, existe una representación no literal para éste tipo de datos. No obstante, las literales de cadenas de caracteres pueden ser asignadas a campos MEMO con REPLACE.

La siguiente tabla da una sinópsis de todas las operaciones en lenguaje Clipper que son válidas para los tipos de datos MEMO. Las operaciones listadas aquí son las designadas para trabajar específicamente con campos MEMO, pero ellas pueden además ser usadas con cadenas de caracteres largas.

Tabla 1-2: Operaciones MEMO

Operación	Descripción
HARDCR()	Reemplaza el carriage/return blando por duro
MEMOEDIT()	Edita contenido
MEMOLINE()	Extrae una línea de texto
MEMOREAD()	Lee desde un fichero de disco
MEMOTRAN()	Reemplaza los returns blandos y duros
MEMOWRIT()	Escribe hacia un fichero en el disco
MLCOUNT()	Contador de líneas
MLPOS()	Calcula posición

Además de estas operaciones especializadas con campos MEMO, todas las operaciones de caracteres listadas anteriormente, pueden ser usadas con los campos MEMO.

5.1.3.- Fecha

El tipo de dato fecha es usado para identificar los items de dato que representan las fechas del calendario. En Clipper, Ud. puede manipular fechas en diferentes formatos, por ejemplo, la búsqueda del número de dias entre dos fechas y la determinación de que día va a ser 10 dias después de ahora. El conjunto de caracteres fecha, esta definido con los dígitos que van del 0-9 y un caracter especificado como separador por SET DATE.

En Clipper 5.0 el comando SET EPOCH ha sido adicionado al lenguaje para cambiar el siglo asumido por defecto. Vea la entrada para este comando en la sección de comandos standard para más información.

Las fechas son formadas por un encadenamiento de dígitos y separadores en un orden particular, en dependencia del valor del actual SET DATE. Por defecto, SET DATE es AMERICAN y las fechas son de la forma "mm/dd/[aa]aa": mm representa el mes y debe ser un valor entre 1 y 12; dd representa el dia el cual debe estar dentro de 1 y 31; [aa], si es especificado representa el siglo y debe estar entre 0 y 29; aa representa el año el cual debe estar entre 0 y 99; y el / que es el separador.

Clipper soporta todas las fechas válidas en el rango de 01/01/0100 hasta 12/31/2999 así como un nulo, o blanco fecha. Ud. puede especificar el siglo como parte de una literal de fecha sin necesidad del estado de SET CENTURY; no obstante, Ud. no puede dar entrada a una fecha que no pertenece al siglo XX en una variable @ ... GET, ni podrá tampoco visualizar el siglo a menos que SET CENTURY esté en ON.

Las fechas no pueden tener una franca representación literal parecido a una cadena de caracteres y números. En cambio, Ud. puede usar la función CTOD() para convertir una constante de cadenas de caracteres expresada en forma de fecha hacia un valor de fecha actual. Para hacer esto se forma la fecha de acuerdo a las reglas descritas con anterioridad y se encierra ésta en uno de los pares de delimitadores de cadena de caracteres. Entonces se usa la cadena fecha como argumento de CTOD() como es ilustrado en el ejemplo siguiente:

CTOD("1/15/89")

CTOD('03/5/63')

CTOD([09/28/1693])

CTOD("01/01/0100")

CTOD() chequea estos argumentos para asegurarse que esta es una fecha válida. Por ejemplo, "11/31/89" y "02/29/90" serán fechas no válidas, porque Noviembre tiene solo 30 dias y 1990 no es un año bisiesto. CTOD() convierte una fecha inválida en una fecha nula.

Nota: Para expresar un blanco o fecha nula, use una cadena nula como argumento para CTOD() (por ejemplo, CTOD("")).

La tabla a continuación proporciona una sinópsis de todas las operaciones en el lenguaje Clipper que son válidas para el tipo de datos fecha. Estas operaciones actúan sobre uno o más expresiones de fecha para producir un resultado. El tipo de dato del resultado no es necesariamente de tipo fecha.

Tabla 1-3: Operaciones de Fecha

Operación	Descripción
+	Adiciona un número a una fecha
-	Sustrae desde una fecha
++	Incrementa
--	Decrementa
= o ==	Comparar por igualdad
!=, <>, #	Comparar por desigualdad
<	Compara precedencia
<=	Compara precedencia o igual a
>	Compara posterioridad
>=	Compara posterioridad o igual a
= ó STORE	Asignación
:=	Asignación en línea
+=	Adiciona en línea y asigna
-=	Resta en línea y asigna
REPLACE	Remplaza el valor del campo
CDOW()	Calcula el nombre del dia de la semana
CMONTH()	Calcula el nombre del mes
DAY()	Extrae número del dia
DESCEND()	Convierte a forma complementada
DOW()	Calcula el dia del número de la semana
DTOC()	Conviete a cadena según SET DATE
DTOS()	Convierte a caracter en formato ordenado
EMPTY()	Comprueba para nulo
MONTH()	Extrae el número del mes
TRANSFORM()	Convierte a formato de cadena
TYPE()	Evalúa el tipo de dato (macrosustitución)
VALTYPE()	Evalúa el tipo de dato directamente
YEAR()	Extrae el número del año completamente

Las fechas se comparan de acuerdo a su orden cronológico.

5.1.4.- Numérico

El tipo de dato numérico es usado para identificar un item de dato que Ud. desea manipular matemáticamente (por ejemplo, la ejecución de una suma, multiplicación, y otras funciones matemáticas). El conjunto de caracteres numéricos esta definido entre los dígitos del 0-9, el punto para representar el punto decimal y los signos + y - para indicar el signo del número.

Con la excepción de los campos, Clipper 5.0 almacena valores numéricos usando el formato standard de punto flotante de doble precisión, siendo el rango de los números de 10^-308 a 10^308. La precisión numérica es garantizada hasta 16 dígitos significativos y el formateo de un valor numérico por el display es garantizado a una longitud de 32 (es decir, 30 dígitos un signo y un punto decimal). Esto significa que un número mayor de 32 bytes va a ser visualizado como asteriscos, y los otros dígitos después de los 16 más significativos, son mostrados como ceros.

Cuando un valor es almacenado en un campo numérico de un fichero de base de datos (.dbf) este es convertido de formato IEEE a una representación mostrable. Cuando un campo numérico es actualizado en un fichero, este es reconvertido a formato IEEE antes de que cualquier operación sea efectuada en él. Debido a que el formato mostrable de un valor numérico es garantizado hasta una longitud de 32 bytes, este es el largo recomendado para la longitud de un campo numérico.

Las literales numéricas estan formadas por un encadenamiento de uno o más de los siguientes símbolos: un signo más o menos encabezando, uno o más dígitos que representan la porción entera del número, un punto decimal y uno o más dígitos para representar la parte fraccionaria del número. Algunas constantes numéricas válidas se muestran a continuación:

- 1234
- 1234.5678
- -1234
- +1234.5678
- -.5678

Nota: A diferencia de las cadenas de caracteres los valores de las literales numéricas no son delimitadas. Si Ud. encierra un número -o cualquier cadena de caracteres- entre delimitadores, este llegará a ser una cadena de caracteres.

La siguiente tabla da una sinópsis de todas las operaciones en el lenguaje Clipper para un tipo de dato numérico. Estas operaciones actúan sobre una o más expresiones numéricas para producir un resultado. El tipo de dato resultante no tiene que ser necesariamente numérico.

Tabla 1-4: Operaciones numéricas

Operación	Descripción
+	Adición o unario positivo
-	Sustracción o unario negativo
*	Multiplicación
/	División

%	Módulo
++	Incremento
--	Decremento
= ó ==	Compara si igual
!=, <>, #	Compara si desigual
<	Compara si menor que
<=	Compara si menor o igual que
>	Compara si mayor que
>=	Compara si mayor o igual que
= ó STORE	Asignación
:=	Asignación en línea
+=	Adición en línea y asignación
-=	Sustracción en línea y asignación
*=	Multiplicación en línea y asignación
/=	División en línea y asignación
**=	Exponenciación en línea y asignación
REPLACE	Reemplaza valor en campo
ABS()	Calcula valor absoluto
CHR()	Convierte a caacter ASCII equivalente
DESCEND()	Convierte a forma complementada
EMPTY()	Comprueba si cero
EXP()	Exponenciación con e como base
INT()	Convierte a entero
LOG()	Calcula logaritmo natural
MAX()	Calcula el máximo
MIN()	Calcula el mínimo
MOD()	Calcula el módulo dBASE III Plus
ROUND()	Redondea
SQRT()	Calcula raíz cuadrada
STR()	Convierte a caracter
TRANSFORM()	Convierte a cadena formateada
TYPE()	Evalúa el tipo de dato (macro-sustitución)
VALTYPE()	Evalúa tipo de dato directamente

Los números son comparados acorde a sus actuales valores numéricos. Note que el operador = y el operador == se comportan idénticamente cuando se comparan números -ambos chequean igualdad con el máximo de recisión permitida por formato de punto flotante de 8 bytes de la IEEE.

Las operaciones numéricas -incluyendo las comparaciones- son completadas sin afectarse por algún comando SET. Si una operación numérica o de comparación produjera resultados no esperados esto es probablemente debido al formateo de display automático o por el valor de punto flotante. El formateo del display es afectado por los comandos SET FIXED y SET DECIMALS.

5.1.5.- Lógico

El tipo de dato lógico es usado para identificar a un item de datos que es booleano por naturaleza. Un item de dato lógico típicamente son esos valores de verdadero o falso, sí o no, y ON u OFF. En Clipper el juego de caracteres lógicos consiste de las letras Y, y, T, t, N, n, F y f.

Por definición solo dos valores lógicos son posibles pero en Clipper hay diversos modos de representarlos. Para formar una literal de valor lógico se encierra uno de los caracteres en el juego de caracteres lógicos definidos entre dos períodos (.). Los períodos son delimitadores de valores lógicos justamente como una cadena de caracteres entre comillas.

El valor literal .y., .Y., .t. y .T. representan verdadero. El valor literal .N., .N., .f. y .F. representan falso. La representación literal preferida es .T. y .F.

La siguiente tabla da una sinópsis de todos los operadores del lenguaje Clipper que son válidos para este tipo de datos. Estas operaciones actúan en una o más expresiones lógicas para producir un resultado. El tipo de dato resultante no es necesariamente lógico.

Tabla 1-5: Operaciones lógicas

Operación	Descripción
= ó ==	Compara si igualdad
!=, <>, #	Compara si desigualdad

Nota: Para el propósito de comparación de valores lógicos, falso (.F.) es siempre menor que verdadero (.T.).

5.1.6.- NIL

NIL es un nuevo tipo de dato introducido en Clipper 5.0 que siempre usted manipulará en variables no inicializadas sin generar un error de ejecución. Este es un tipo de dato que tiene un solo valor posible, el valor NIL.

La representación literal para NIL es la cadena de caracteres "NIL" sin delimitadores. Cuando es referenciado por un comando o función a display, el valor NIL es mostrado. Note además que NIL es una palabra reservada en Clipper 5.0.

La siguiente tabla da una sinópsis de todas las operaciones en lenguaje Clipper que son válidas para el tipo de datos NIL. Excepto para estas operaciones, el intento de operar con un NIL provoca un error en tiempo de ejecución. Por ejemplo, usted no puede reemplazar un campo con NIL, no puede concatenarle NIL a una cadena de caracteres, etc.

Tabla 1-6: Operaciones con NIL

Operación	Descripción
= ó ==	Compara si igual
!=, <>, #	Compara si desigual
<	Compara si menor que
<=	Compara si menor o igual que
>	Compara si mayor que
>=	Compara si mayor o igual que
= ó STORE	Asignación a una variable no campo
:=	Asignación en línea a
= ó ==	Compara si igual
!=, <>, #	Compara si desigual
<	Compara si menor que
<=	Compara si menor o igual que
>	Compara si mayor que
>=	Compara si mayor o igual que
= ó STORE	Asignación a una variable no campo
:=	Asignación en línea a una variable no campo
EMPTY()	Comprueba si NIL
TYPE()	Evalúa el tipo de dato (macro-sustitución)
VALTYPE()	Evalúa el tipo de dato directamente

Para el propósito de comparación, NIL es el único valor que es igual a NIL. Todos los otros valores son mayores que NIL
Descripción

DECLARE	Asigna NIL a elementos de arreglos
LOCAL	Asigna NIL a var.loc. y elementos arreglos
PARAMETERS	Asigna NIL a parámetros por omisión
PRIVATE	Asigna NIL a var.priv. y elem. de arreglos
PUBLIC	Asigna NIL a elementos de arreglos públicos
STATIC	Asigna NIL a var.estáticas y elem.de arreglos

Todas las variables con exepción de las PUBLIC y FIELD son inicializadas con NIL cuando son declaradas o creadas sin inicializarse. Las variables PUBLIC son inicializadas con .F. cuando son creadas y las variables campo nunca pueden contener valor NIL. Cuando un arreglo es creado con una sentencia de declaración, incluyendo PUBLIC, todos los elementos son inicializados con NIL.

Cuando una función o procedimiento es invocado, si un argumento es omitido por permitirse un lugar vacio seguido por coma o por permitirse la omisión al final, un valor NIL es pasado para los mismos. Note que la función o procedimiento no puede distinguir entre un valor NIL que es pasado explícitamente (por ejemplo, una expresión en la lista de argumentos es evaluada como NIL) y uno que es pasado como resultado de la omisión de un argumento.

6.- Operadores

A lo largo de las funciones, constantes y variables, los operadores han sido la construcción básica de bloques de expresiones. Un operador es semejante a una función, el cual efectúa una operación específica y retorna un valor. Esta sección ofrece una discusión general de los operadores, categoriza y describe todos los operadores habilitados en lenguaje el Clipper.

6.1.- Definiciones

Existen diversos términos que son usados en las discusiones de operadores, descritos resumidamente en los siguientes párrafos, en orden, para ayudar a su comprensión en el material restante de esta sección.

6.1.1.- Sobrecargando (Overloading)

En Clipper, ciertos operadores tienen diferentes significados dependiendo del tipo de dato del operando. Por ejemplo, el operador menos (-) puede ser usado para concatenar cadenas de caracteres, así como para sustraer valores numéricos y de fecha. El uso del mismo signo u operador, para diferentes operaciones se denomina "overloading".

6.1.2.- Operadores unarios y binarios

Todos los operadores en Clipper requieren de uno o dos argumentos denominados operandos. Los que requieren un solo operando son llamados operadores unarios, y los que requieren dos operadores son llamados operadores binarios.

Los operadores binarios usan notación "infix" (encajada) que significa que el operador es situado entre operandos. Los operadores unarios usan notación "prefix" o "postfix" donde el operador es situado antes o después del operando, dependiendo del operador y de cómo Ud. quiere usarlo.

Un ejemplo de operador prefix unario es el operador de negación lógica (!):

```
lTrue := .T.
? !lTrue       // Result: .F.
```

El operador de post-incremento es un ejemplo de operador postfix unario. Este operador incrementa al operando por un valor de uno:

```
nCount := 1
nCount++       // Result: nCount is now 2
```

El operador de multiplicación (*) es un ejemplo de operador binario que demuestra notación infix:

```
? 12 * 12      // Result: 144
```

6.1.3.- Precedencia

Las reglas de precedencia determinan el orden en el cual los diferentes operadores son evaluados dentro de una expresión. Estas reglas definen la jerarquía de todos los operadores dentro de una expresión. Los paréntesis pueden ser usados para anular el orden de evaluación por omisión y hacer una expresión complicada más entendible.

6.2.- Manipulación de errores

Los operadores son usados para formar expresiones y deben, por lo tanto, ajustarse a ciertas reglas. Por ejemplo, Ud. no puede usar un operador unario que permita solo notación prefix, como operador postfix y, excepto en pocas y bien definidas circunstancias, Ud. no puede usar operadores binarios en operandos con diferentes tipos de datos. Todas las reglas que se aplican al uso de operadores son descritas en esta sección, y el uso de cualquier operador incorrectamente resulta un error de ejecución.

6.3.- Operadores de cadena

La siguiente tabla lista cada operador de cadena y los cálculos que estos efectúan. Estos operadores son binarios, requiriendo dos operandos de tipo caracter y/o MEMO, y retorna un valor de caracter.

Tabla 1-8: Operadores de cadena

Símbolo	Operación
+	Concatenación
-	Concatenación sin intervención de espacios

El operador concatenación (-) mueve los espacios al final del primer operando hacia el final de la cadena resultante, no interviniendo, de este modo, los espacios entre las dos cadenas originales. El operador + deja los espacios intactos.

6.4.- Operadores de fecha

Los operadores matemáticos + y - debatidos en la próxima sección pueden ser usados para adicionar o sustraer un número de dias a un valor de fecha. El resultado de semejante operación es una fecha con un número de dias pasados (sustracción) o futuros (adición). El orden de los operandos en el caso de la sustracción es significativo -el valor de fecha debe venir primero. Esta es una excepción de la regla que no permite operaciones usando tipos de datos mezclados. En la regla precedente, definida al final de esta sección, este tipo de operación mezclada es considerada como matemática mejor que de fecha.

Ud. puede además sustraerle una fecha a cualquier otra usando el operador matemático de sustracción. El resultado de este tipo de operación es un valor numérico que representa el número de dias entre las dos fechas. La sustracción de una fecha por otra, es el único operador de fecha verdadero.

6.5.- Operadores matemáticos

La siguiente tabla lista cada operador matemático y los cálculos que estos permiten. Como regla general, estos operadores requieren operandos de tipo numérico (vea Operadores de fecha expuestos anteriormente para conocer las excepciones) y retorna un valor numérico. Excepto donde es indicado en la tabla, los operadores matemáticos son binarios. El operador unario usa notación prefix.

Tabla 1-9: Operadores matemáticos

Símbolo	Operación
+	Adición o unario positivo
-	Sustracción o unario negativo
*	Multiplicación
/	División
%	Módulo
** ó ^	Exponenciación

El operador módulo retorna el resto de una operacion de división. Si lo que se desea como resultado de una división es el resto, use el operador módulo (%) en vez del operador de división (/).

6.6.- Operadores relacionales

Abajo se listan los operadores relacionales y sus propósitos. Todos ellos son operadores binarios requiriendo ambos operandos del mismo tipo de dato o al menos un operando NIL. El resultado de una operación relacional es un valor lógico.

Tabla 1-10: Operadores relacionales

Símbolo	Operación
<	menor que
>	mayor que
=	igual a
==	exactamente igual para caracteres; igual para otros tipos de datos
<>, #, !=	no igual
<=	menor o igual a
>=	mayor o igual a
$	si esta contenido en el conjunto; o si es subconjunto de

SET EXACT afecta a los operadores de $ y de = cuando se comparan cadenas. Si SET EXACT esta en ON, ambas cadenas deben ser idénticas en contenido y longitud (excepto en los espacios al final) para que estas operaciones retornen verdadero (.T.).

El operador == compara cadenas (es decir, tipos de caracteres y MEMO) para una igualdad exacta en longitud y contenido, incluyendo los espacios al final. Para los arreglos, este chequea que ambos arreglos refieran a la misma área de memoria. Para el resto de los tipos de datos, el operador == es equivalente al operador =.

6.7.- Operadores lógicos

La siguiente tabla contiene la lista de operadores lógicos.

Tabla 1-11: Operadores lógicos

Símbolo	Operación
.AND.	And lógico
.OR.	Or lógico
.NOT. ó !	Negación lógica

Todos los operadores lógicos requieren operandos lógicos y, excepto para la negación ellos son binarios. .NOT. y ! son operadores prefix unario. El resultado de una operación lógica es siempre un valor lógico.

El camino más rápido para definir estos operadores es saber cuando ellos retornan verdadero (.T.): .AND. retorna verdadero (.T.) si ambos operadores son verdaderos; .OR. retorna verdadero si uno de los operandos es verdadero; y .NOT. retorna (.T.) si sus operandos son falsos. Las tablas de verdad definen todos los valores lógicos y son mostrados a continuación:

Tabla 1-12: Tabla de verdad

ÚÚ

.AND.	.T.	.F. °	.OR.	.T.	.F. °	.NOT.	.T.	.F.
éé								
.T.	.T.	.F. °	.T.	.T.	.T. °		.F.	.T.

.F. .F. .F. ° .F. .T. .F. °
ÙÙÙ

6.8.- Operadores de asignación

Clipper tienen varios operadores que asignan valores a variables que son resumidos en la siguiente tabla:

Tabla 1-13: Operadores de asignación

Símbolo	Operación
=	Asignación
:=	Asignación en línea
+=	Adición en línea (o concatenación) y asigna
-=	Sustrae en línea (o concatenación) y asigna
*=	Multiplicación en línea
/=	División en línea
**= ó ^=	Exponenciación en línea
%=	Módulo en línea

Todos los operadores de asignación son binarios y requieren una variable simple como primer operando. El segundo operando puede ser cualquier expresión válida, incluyendo el NIL, con tal que el tipo de dato sea conforme a la operación. Para los operadores compuestos (es decir, todos excepto = y :=) la variable usada como primer operando debe existir y debe ser del mismo tipo de dato que el segundo operando, excepto para algunos casos de operaciones con fecha y número. Estas excepciones son descritas en la siguiente tabla la cual resumen los operadores de asignación por tipos de datos.

Tabla 1-14: Operadores de asignación por tipo de dato

Símbolo	Operación
=	Todos
:=	Todos
+=	Caracter, fecha, memo, numérico
-=	Caracter, fecha, memo, numérico
*=	Numérico
/=	Numérico
**= ó ^=	Numérico
%=	Numérico

Además de esos tipo de datos, -= y += permiten como primer operando una expresión de fecha y como segundo operando una expresión numérica. El resultado es que un número de dias es adicionado o sustraído a una fecha, asignándole el nuevo valor.

Nota: Cualquier operador de asignación asume MEMVAR si la referencia a la variable es ambigua. Si se intenta hacer una asignación hacia un campo, la variable campo debe estar declarada con una sentencia FIELD o referenciada por un alias ambos con el alias asignado o el FIELD -> alias.

6.8.1.- Asignación simple

El operador de asignación simple, =, es usado para asignar un valor a una variable. Este opera idénticamente igual al comando STORE al inicializar una simple variable, y debe ser especificado como una sentencia de programa. Si es usado dentro de una expresión es interpretado como un operador de igualdad. Por ejemplo:

```
nValue = 25
nNewValue = SQRT( nValue ) ** 5
nOldValue = nValue
```

Todas son sentencias válidas de asignación simple. El primer operando, como con todas las demás sentencias de asignación, puede ser un nombre de campo, así como cualquier otra variable. El resultado es que el campo es reemplazado por un nuevo valor. Por ejemplo, las siguientes dos líneas de código ejecutan exactamente la misma función de reemplazamiento del valor actual del campo CustAge con el valor numérico 20:

```
FIELD->CustAge = 20
REPLACE CustAge WITH 20
```

6.8.2.- Asignación en línea

El operador de asignación en línea, :=, puede ser usado alternativamente con el operador de simple asignación. Por ejemplo:

```
nValue := 25
nNewValue := SQRT( nValue ) ** 5
nOldValue := nValue
```

Este operador, como quiera que sea, es un operador en línea cuyo significado es que puede ser usado en cualquier comando o función cuya sintaxis permita el uso de una expresión. El tipo de dato en una operación de asignación en linea es igual al tipo de dato del segundo operando. Ejemplos:

```
LOCAL nValues := 10
//
IF ( dDate := (DATE() - 1000 )) = CTOD ("12/20/79")
//
?SQRT(nValue := (nValue ** 2))
//
cTransNo := cSortNo := (Custid + DTOC(DATE()))
```

Este último ejemplo demuestra como ejecutar una múltiple asignación usando el operador de asignación en línea muy similar a como usted lo pudiera hacer con el comando STORE. Cuando := es usado en esta forma las asignaciones son ejecutadas de derecha a izquierda. Esta característica es particularmente muy provechosa cuando usted necesita almacenar el mismo valor a muchos campos diferentes, posiblemente en diferentes ficheros de base de datos:

```
Custfile->CustId := TransFile->TransNo := ( CustId + DTOC(DATE())
```

6.8.3.- Asignaciones Compuestas

Los operadores de asignación compuesta ejecutan una operación entre los dos operandos y entonces asignan el resultado hacia el primer operando. Estos operadores son resumidos y definidos en la siguiente tabla.

Tabla 1-15: Definición de operadores de asignación compuesta.

Símbolo	Ejemplo	Definición
+=	a += b	a := (a+b)
-=	a -= b	a := (a-b)

*=	a *= b	a := (a*b)
/=	a /= b	a := (a/b)
%=	a %= b	a := (a%b)
**= ó ^=	a ^= b	a := (a^b)

Note que las definiciones para estos operadores usan el operador de asignación en línea. Esto significa que todos los operadores compuestos son también operadores de asignación en línea que pueden ser usados como tales. El tipo de dato para estas operaciones es determinado por el segundo operando usando las definiciones de la tabla anterior. Si una operación de asignación es formada correctamente, esta podrá ser el tipo de dato del primer operando en la sentencia original de asignación compuesta.

Nota: No es aconsejable usar los operadores de asignación con REPLACE o UPDATE ya que la clausula WITH realiza la operación de asignación como parte de la sintaxis del comando.

6.9.-Operadores de Incremento y Decremento.

Los operadores de incremento (++) y decremento (--), nuevos en Clipper 5.0, son resumidos en la siguiente tabla:

Tabla 1-16 Operadores de incremento y decremento

Simbolo	Operación
++	Prefix o postfix de incremento
--	Prefix o postfix de decremento

Ambos operadores son unarios, pueden ser usados en un operando numérico o de fecha. A diferencia de otros operandos los cuales operan sobre expresiones más complicadas, el operando aquí debe ser sencillo, y no variable de campo. El tipo de dato resultante es del mismo tipo del operando.

EL operador ++ incrementa, o aumenta el valor del operando en uno, y el operador -- decrementa o disminuye el valor del operando en uno. Por consiguiente, ambos operadores realizan una operación, así como una asignación sobre y hacia el operando. A continuación se muestra como estos operadores pueden ser definidos en términos de una adición (o substracción) y un operador de asignación:

* value ++ es equivalente a value := value + 1
* value -- es equivalente a value := value - 1

Ambos operadores pueden ser especificados como prefix o postfix: La forma de prefix cambia el valor del operando antes de que la asignación sea hecha, mientras que la forma postfix cambia el valor después. En otras palabras, la forma postfix demora la asignación de la operación hasta que el resto de la expresión sea evaluada y la forma prefix realiza la asignación antes que cualquier operación en la expresión.

Los siguientes códigos a continuación ilustran el operador preincremento en una sentencia de asignación:

```
nValue := 0
nNewValue := ++nValue
? nNewValue                 // resultado 1
? nValue                    // resultado 1
```

Ya que el incremento ocurre antes de que la asignación sea hecha, ambas variables tiene el mismo valor. El próximo ejemplo demuestra el operador postdecremento:

```
nValue := 1
```

```
nNewValue := nValue --
? nNewValue              // resultado 1
? nValue                 // resultado 0
```

Debido a que la asignación es hecha antes, de que la variable original sea decrementada, las dos variables no tienen el mismo valor. El próximo ejemplo ilustra la diferencia en la forma prefix y postfix de estos operadores.

```
nValue := 10
? nValue ++ * nValue        // resultado 110
? nValue                    // resultado 11
```

Aquí el operador postincremento es usado para incrementar el primer operando de la operación de multiplicación en uno, haciendo el valor 11;de cualquier modo, la asignación de este nuevo valor a la variable nValue no se va a realizar hasta que la expresión sea evaluada completamente. Por consiguiente este valor se mantiene con 10 cuando la operación de multiplicación ocurre y el resultado de 10 * 11 es 110. Finalmente, cuando nValue es requerida otra vez después que la expresión es evaluada, la asignación de postincremento refleja un nuevo valor, 11.

En el próximo ejemplo, el operador predecremento es usado para decrementar el primer operando de la multiplicación en uno, haciendo este valor 9. Adicionalmente la asignación de este nuevo valor hacia la variable nValue tiene lugar antes que el resto de la expresión sea evaluada. Por consiguiente, el nuevo valor es usado para ejecutar la operación de multiplicación y el resultado de 9 * 9 es 81:

```
nValue := 10
? --nValue  * nValue        // resultado 81
? nValue                    // resultado 9
```

6.10 Operadores especiales

La siguiente tabla muestra todos los otros símbolos que tienen especial significado en el lenguaje Clipper. Ellos son operadores especiales que regularmente aparecen en las expresiones.

Tabla 1-17 Operadores especiales

Simbolo	Operación
()	Función o agrupamiento
[]	Elemento de arreglo
{}	Definición de un arreglo constante
->	Identificador del alias
&	Compila y ejecuta
@	Pase por referencia

Los () son usados en una expresión para operaciones en grupos y forzar la evaluación en un orden particular o para indicar una llamada a función. Cuando el operador especifica un grupo, lo que está dentro del () debe ser una expresión válida. Subexpresiones adicionales pueden ser agrupadas. Para la llamada a función, un nombre válido debe preceder a un '(' y la lista de argumentos si existieran, seguida de un ')'.

El operador subíndice [] es usado para referenciar un elemento de un arreglo. El nombre del arreglo previamente declarado debe preceder a un '[' y el índice del elemento del arreglo debe aparecer como una expresión numérica, terminando con un ']'.

Las {} son usadas para crear una referencia a un arreglo literal. Los elementos del arreglo deben estar dentro de las {} y separados por coma. Las {} también son usadas para crear bloques de código. Los argumentos de los bloques de código estan adicionalmente delimitados por ||, y las expresiones definidas dentro de los bloques de código son separadas por comas.

El identificador alias (->) es usado para hacer una referencia explícita a un campo o variable. Si el nombre de un alias precede al operador, este debe ser seguido por un nombre de campo de una base de datos y por cualquier expresión válida entre (). Adicionalmente las palabras claves FIELD y MEMVAR pueden preceder al operador seguido por un campo válido o un identificador de variable.

El símbolo ampersand (&) es referido al operador de compilación y ejecución. Es un operador unario donde el único operando válido es una variable de caracteres.

El operador paso por referencia (@) es válido únicamente en una lista de argumentos de una llamada a función o procedimiento usando la sintáxis de llamada a función. Es un operador prefix unario en el cual el operando puede ser cualquier nombre de variable. Trabaja forzando al argumento a ser pasado por referencia, en vez de por valor, a la función o procedimiento.

6.11.- Precedencia de un Operador

Cuando una expresión es evaluada con dos o más operaciones que no están explícitamente agrupadas entre paréntesis, Clipper usa un conjunto de reglas establecidas que determinan el orden en el cual los operadores serán evaluados. Estas reglas, llamadas reglas de precedencias, definen la jerarquía de todos los operadores tratados hasta ahora en esta sección.

Nota: Todas las llamadas a funciones y otros operadores especiales en una expresión son evaluados antes de cualquier otro operador.

6.11.1.- Categorías de precedencias

Generalmente las expresiones en Clipper son formadas usando operaciones que manipulan un solo tipo de dato. Por ejemplo, una expresión puede concatenar diversas cadenas de caracteres o efectuar una serie de operaciones matemáticas de diversos números. Existe, por lo tanto, expresiones en las cuales es necesaria la evaluación de los diferentes tipos de operaciones. Por ejemplo, una expresión lógica compleja puede implicar diversas operaciones relativas en diferentes tipos de datos que son conectados con operadores lógicos como se muestra en este ejemplo:

```
cString1 $ cString2 .AND. nVal1++ > nVal2 * 10
```

Cuando más de un tipo de operador aparece en una expresión, todas las subexpresiones son evaluadas por cada nivel de precedencia antes de que las subexpresiones del próximo nivel sean evaluadas. Exepto para la múltiple asignación en línea, todas las operaciones a cada nivel son efectuadas en orden de izquierda a derecha; múltiples asignaciones en línea son ejecutadas de izquierda a derecha. El orden de precedencia para los operadores, por categoría, es la que sigue:

1. Pre-incrementador y pre-decrementador.
2. Cadena
3. Fecha
4. Matemático
5. Relacional
6. Lógico
7. De asignación
8. Post-incremento y post-decremento

La parte no asignada de los operadores de asignación compuestos (por ejemplo, la parte multiplicación de *=) existe en los niveles 2, 3 o 4 dependiendo de la operación, y la porción de asignación existe en el nivel 7.

6.11.2.- Precedencia dentro de una categoría.

Dentro de cada categoría, los operadores individuales también tienen una precedencia establecida, la cual es significativa, especialmente en las operaciones matemáticas. Observe las siguientes expresiones:

* 5 * 10 + 6 / 2 * 6 / 2 + 5 * 10

Algebraicamente, estas expresiones son equivalentes, pero son escritas diferente el resultado de las dos expresiones es 53. Si Clipper no hubiera establecido un orden de precedencia para evaluar los operadores matemáticos, usted no evaluaría esta expresión correctamente sin el uso de paréntesis.

Sin precedencia entre los operadores, todos los operadores matemáticos serían ejecutados en orden de izquierda a derecha, y las dos formas de expresión mostradas anteriormente serían evaluadas como sigue: 5 * 10 = 50, 50 + 6 = 56 , 56 / 2 = 28; la segunda sería 6 / 2 = 3, 3 + 5 = 8, 8 * 10 = 80. Ninguno de los resultados es correcto, y ambos resultados no son iguales.

Para salvarlo a usted del uso de paréntesis en expresiones complicadas y para asegurar que expresiones equivalentes den el mismo resultado, Clipper usa un orden establecido de evaluación de operaciones en cada categoría.

6.11.3.- Pre-incremento y Pre-decremento

Como se mencionó anteriormente las formas prefix y postfix de los operadores de incremento y decremento son considerados como categorías separadas ya que ellos tienen distintos niveles de precedencia. Estas categorías se refieren a la forma prefix de los operadores incremento y decremento.

Ambos operadores en esta categoría (++ y --) existen con el mismo nivel de precedencia y son ejecutados en orden de izquierda a derecha.

6.11.4.- Cadena

Los operadores de concatenación de cadenas (+ y -) existen al mismo nivel de precedencia y son ejecutados de izquierda a derecha.

6.11.5.- Fecha

El único operador de dato verdadero es el de sustracción (-) de una fecha con otra. Las sustracciones de fecha son ejecutadas de izquierda a derecha.

6.11.6.- Matemáticos

Cuando más de un operador matemático aparece en una expresión, todas las subexpresiones son evaluadas para cada nivel de precedencia antes que las subexpresiones del próximo nivel sean evaluadas. Todas las operaciones en cada nivel son ejecutadas de izquierda a derecha. El orden de precedencia de los operadores matemáticos es como sigue:

1. Unario positivo y negativo (+, -)
2. Exponenciación (** , ^)
3. Multiplicación, división y módulo (*, /, %)
4. Adición y sustracción (+, -)

6.11.7.- Relacional

Todos los operadores relacionales existen con el mismo nivel de precedencia y son ejecutados de izquierda a derecha.

6.11.8.- Lógicos

Semejante a los operadores matemáticos, los operadores lógicos tambien tiene establecido un orden de precedencia. Cuando más de un operador lógico aparece en una expresión, todas las subexpresiones son evaluadas para cada nivel de precedencia antes que las subexpresiones del próximo nivel sean evaluadas. Todas las operaciones del mismo nivel son ejecutadas de izquierda a derecha. El orden de precedencia de los operadores lógicos es el que sigue:

1. Unario negativo (.NOT.)
2. And Lógico (.AND.)
3. Or lógico (.OR.)

6.11.9. - Asignación

Todos los operadores de asignación existen con el mismo nivel de precedencia y son ejecutados de derecha a izquierda. Para los operadores compuestos la parte no asignada (por ejemplo, adición o concatenación) de la operación es ejecutaa primero, seguida inmediatamente por la asignación.

Las operaciones de asignación de incremento y decremento existen en su propio nivel de precedencia, y no son parte de la categoría de asignación.

6.11.10.- Post-incremento y Post-decremento

Ambos operadores en esta categoría (++ y --) existen en el mismo nivel de precedencia y son ejecutados de izquierda a derecha.

6.11.11.- Paréntesis

El orden de precedencia de evaluación de expresiones puede ser alterado usando paréntesis. Cuando los paréntesis se presentan en una expresión todas las subexpresiones dentro de los mismos son evaluadas primero sando las reglas de precedencia descritas en esta sección, si es necesario. Si los paréntesis son anidados, la evaluación es ejecutada comenzando por el paréntesis más interno y procede hacia afuera.

Note que aunque el lenguaje Clipper provee un orden específico de precedencia para la evaluación de expresiones, ésta es la mejor práctica de programación para las operaciones en grupo, ya que se logra una lectura más amena.

7.- Macro Operador (&)

El macro operador en Clipper 5.0 es un operador especial que permite una compilación en tiempo de ejecución de las expresiones y la sustitución de textos dentro de cadenas. Donde quiera que el macro operador (&) es encontrado, el operando es sometido a una compilación especial en tiempo de ejecución, a través del compilador de macros que puede compilar expresiones, pero no sentencias ni comandos.

Una macro puede ser especificada usando una variable que contenga una cadena de caracteres o una expresión que retorne una cadena de caracteres. Si una variable de caracteres es usada para especificar una macro, esta será referida como una macro variable y especificada de este modo:

& < macroVar >

El símbolo período (.) es el macro terminador y es usado para indicar el fin de la macro variable y puede distinguir a la macro variable en un texto adyacente dentro de la sentencia.

El macro operador (&) puede además ser aplicado a una expresión referida como una macro expresión, si la expresión evalúa a un valor de carateres y es encerrada entre paréntesis, como aquí:

& (< macroExp >)

Además la expresión es evaluada primero y la macro operación es ejecutada con el valor de caracter resultante. Esto permite que el contenido de campos y elementos de arreglos puedan ser compilados y ejecutados.

El macro operador (&) permite la sustitución de texto o compilación en tiempo de ejecución de una expresión, dependiendo de como esta es especificada.

7.1.- Sustitución de textos

Cuando una referencia a una macro variable privada o pública insertada dentro de una cadena de caracteres es encontrada, como es:

```
cMacro := "por aquí"
? "Buenas &cMacro"          // resultado Buenaspor aquí
```

El contenido de la macrovariable es sustituido por el de la variable de referencia. Si una macro expresión es especificada

por ejemplo, &(macro1 + macro2),

y la macro variable resultante de la evaluación de esta expresión, es local, estática, variable campo, o un elemento de arreglo, ésta es tratada como una literal de texto y no es expandida.

7.2.- Compilación y ejecución

Cuando una macro-variable o macro-expresión especificada dentro de una expresión es encontrada, ésta es tratada como una expresión y el macro-símbolo (&) se comporta como un operador de compilación y ejecución. Si la macro, es especificada sobre una macro-variable como ésta:

```
cMacro := "DTOC(DATE())"
? &cMacro
```

El contenido de la macro-variable es compilado por el macro compilador y entonces ejecutado. El código compilado es descartado. Si una expresión es especificada y encerrada entre paréntesis y precedida por el macro operador (&) como ésta:

```
? &(INDEXKEY(0))
```

la expresión es evaluada y la cadena de caracteres resultante es compilada y ejecutada como si fuera una macro-variable.

Uno de los efectos interesantes de la macro expresión es que usted puede compilar una cadena de caracteres conteniendo una definición de bloque de código, como ésta:

```
bBlock := "{ |exp| QOUT(exp) }"
 .
 .
EVAL(&bBlock, DATE())
```

En este caso, la cadena de caracteres conteniendo el bloque de código es compilada y la ejecución de esta parte de la operación retorna el bloque de código como un valor. El bloque de código resultante puede ser salvado y evaluado más adelante con la función EVAL().

7.3.- Relación (afinidad) con los comandos

Por causa de la larga historia del macro operador (&) en Clipper y sus antece dentes es importante entender con precisión la naturaleza de la relación entre comandos y macros.

7.3.1.- Usándose con palabras claves (comandos)

Primero, el macro operador (&) no puede ser usado para sustituir o compilar, palabras reservadas que sean comandos. En Clipper 5.0, esto no es significativo desde que los comandos son pre- procesados dentro de sentencias y expresiones en tiempo de compilación. La redefinición de palabras reservadas que sean comandos, de cualquier modo, puede ser acompañada por modificaciones a la definición de comandos en STD.CH, actualizando la definición de comandos existentes con nuevas definiciones usando la directiva # command o redefiniendo una palabra reservada que sea comando usando la directiva #translate. En cualquier caso la redefinición de palabras claves que son comandos puede solo ocurrir en tiempo de compilación en vez de en tiempo de ejecución.

7.3.2.- Usándose como argumento de comando

Segundo, en versiones anteriores de Clipper así como en otros dialectos las macro variables fueron frecuentemente usadas para especificar los argumentos de los comandos que requerian valores literales de texto. Esto incluye todos los argumentos de comandos de ficheros así como comandos SET con argumentos conmutables. En esos casos usted puede ahora usar una expresión extendida en lugar de un argumento literal si la expresión es encerrada entre paréntesis. Por ejemplo, el siguiente:

```
xcDatabase = "Invoices"
USE &xcDatabase
```

puede ser reemplazada por:

```
xcDatabase = "Invoices"
USE (xcDatabase)
```

El uso de expresiones extendidas es importante especialmente si usted esta usando variables locales y estáticas. Los comandos son pre-procesados generalmente a través de llamadas a funciones con sus argumentos trasladados dentro de argumentos de función como un valor Clipper legal. Con los comandos de fichero, por ejemplo, los nombres de ficheros son stringifield (sustituye macros dentro de cadenas, por su significado) usando el smart stringify result-marker y pasado como argumento a una función que ejecuta las acciones deseadas. Si una literal o valor macro es especificada como argumento de un comando, éste es stringifield. Si, el argumento es una expresión extendida, el mismo es escrito en el texto resultante exactamente como fue especificado. Por ejemplo, la siguiente especificación del comando RENAME:

```
#command RENAME <xcOld> TO <xcNew>=>FRENAME(<(xcOld)>,<(xcNew)>)
//
RENAME &xcOld TO &xcNew
RENAME (xcOld) TO (xcNew)
```

es escrito hacia el texto resultante como este:

```
FRENAME("&xcOld", "&xcNew")
FRENAME(xcOld, xcNew)
```

cuando es pre-procesado. Cuando la macro variable es stringifield, su nombre es ocultado en la cadena y no es compilado. Más tarde, en tiempo de ejecución, ella es sustituída dentro de la cadena y pasada como argumento a la función FRENAME(). Estas macro variables locales y estáticas son excluídas ya que sus nombres no están presentes en tiempo de ejecución para ser sustituídos. Las variables públicas y privadas, de cualquier modo se comportan como usted pueda esperar. Si esto le puede parecer confuso, refiérase a la sección de variables para más información acerca de las diferencias entre las variables estáticas y locales, y variables privadas y públicas.

Nota: Las expresiones extendidas son denotadas en la sintaxis del comando como metasímbolos precedidos con una letra minúscula x.

7.3.3.- Usándolo con listas

El operador macro no va a sustituir totalmente o tratar una lista como argumento de otros comandos. En particular, estos son comandos donde una lista de argumentos son procesados dentro de un arreglo o un bloque de códigos. Ejemplos de estos argumentos son los de las cláusulas FIELD y SET INDEX. Una excepción es el comando SET COLOR el cual pre-procesa la lista de colores dentro de una cadena de caracteres y pasa éstos hacia la función SETCOLOR().

En cualquier caso, la lista de argumentos podrá siempre ser especificada como expresiones extendidas con cada elemento de la lista especificado:

```
LOCAL xcIndex := {"Ntx1", "Ntx2"}
SET INDEX TO (xcIndex[1]), (xcIndex[2])
```

7.4.- Macros y arreglos

El operador macro puede ser utilizado en combinación con arreglos y elementos de arreglos. De cualquier modo, a causa del incremento de la potencialidad de los arreglos en Clipper 5.0, usted puede encontrar menos necesidad de usar el operador macro (&) para hacer variables de referencias a arreglos. Usted puede ahora asignar las referencias a arreglos, avariables, retornar las referencias a arreglos desde funciones de usuario, y anidar referencias a arreglos dentro de otros arreglos. Además, los arreglos pueden ser creados por la especificación literal de arreglos o usando la función ARRAY(). Para más información sobre arreglos, refiérase a la sección de arreglos de este capítulo.

Las referencias a arreglos, entonces, pueden ser hechas a arreglos y elementos de arreglos usando tanto las macro variables como las macro expresiones, con la restricción que las referencias subscritas no pueden ser hechas en una sentencia PRIVATE o PUBLIC. Existe adicionalmente una restricción y es que el operador (&) no puede ser especificado en una sentencia de declaración tal como una sentencia LOCAL y STATIC. Intentar hacer esto provocará un error fatal del compilador. Por ejemplo, las siguientes demostraciones de referencias a elementos de arreglos usando la macro variable:

```
cName := "aArray"
nElements := 5
cNameElement := "aArray[1]"
PRIVATE &cName.[nElements] // Crea un arreglo con 5 elementos
&cNameElement. := 100      // Asigna 100 al elemento uno
&cName.[3] := "abc"    // Asigna "abc" al elemento 3 del arreglo
```

Un macro-operador (&) puede ser totalmente aplicado a un elemento de arreglo si la referencia es hecha usando una macro expresión. Una referencia a macro variable, de otro modo, puede generar un error en tiempo de ejecución. Por ejemplo, la siguiente lista de valores de todos los campos del artículo actual:

```
USE Customer NEW
aStruct := DBSTRUCT()
FOR nField := 1 TO LENGTH(aStruct)
   ? &(aStruct[nField,1])
NEXT
```

7.4.1.- Macros y bloques de códigos

El macro operador (&) puede ser aplicado a una macro variable o macro expresión en un bloque de código, en muchos casos. Existe una restricción cuando la macro variable o macro expresión contiene una variable declarada. La restricción ocurre cuando una expresión compleja (una expresión que contine un operador y uno o más operandos) incluye el macro operador (&) dentro de un bloque de código. Si esto es intentado, ocurre un error en tiempo de ejecución.

Note que esta restricción tiene importantes implicaciones en el uso de variables estáticas y locales en las cláusulas condicionales de los comandos, ya que estas son blockified y escritas en el texto resultante cuando el comando es preprocesado. Esto se aplica a todas las cláusulas FOR y WHILE, el comando SET FILTER y a la expresión de enlace SET RELATION. En general, todo lo que hay que hacer es recoger la expresión entera dentro de una sola macro variable y aplicarle el macro operador (&) a la variable.

7.4.2.- Usándolo en condiciones de comandos de bases de datos

Cuando se usa el macro operador (&) para especificar cláusulas condicionales de comandos de bases de datos como son las cláusulas FOR o WHILE, existen algunas restricciones basadas en la complejidad de la expresión y el tamaño, como son:

* El tamaño máximo de una cadena que el macro compilador puede procesar es de 254 carcateres.

* Existe un límite a la complejidad de condiciones (la más compleja, el número menor de condiciones que pueden ser especificadas).

7.4.3.- Invocando procedimientos y funciones.

Las llamadas a procedimientos y funciones pueden ser referenciadas usando macro variables y macro expresiones. Con DO, la referencia de macro variable hacia el procedimiento puede incluir todo o parte del nombre del procedimiento. Con una llamada a una función (standard o función usuario), la referencia a macro variable debe incluir el nombre de la función y todos sus argumentos.

Con el advenimiento de los bloques de código en Clipper 5.0, esta no es ya la práctica preferida. En cambio, todas las invocaciones de procedimientos y funciones usando el macro operador pueden ser convertidas en una evaluación de bloques de códigos. Por ejemplo, el siguiente fragmento de código:

```
cProc := "AcctsRpt"
DO &cProc
```

puede ser reemplazada por:

```
bProc := &( "{ || AcctsRpt() }" )
EVAL(bProc)
```

La clara ventaja de un bloque de código en lugar de una macro evaluación es que la compilación de una cadena conteniendo un bloque de código puede ser salvada y por lo tanto debe ser compilada una sola vez. Las macros evaluaciones se compilan cada vez que son referenciadas.

7.4.4.- Referencias externas

Los procedimientos y funciones usuarios que son usados por las macro expresiones y macro variables, pero no estan referenciados en ningún lugar, deben ser declarados externos o el enlazador no los incluirá entonces dentro el fichero ejecutable (.EXE). La única excepción de esta regla es si un procedimiento o función definida por el usuario ha sido pre-enlazada dentro de una biblioteca de pre-enlace (.PLL).

7.4.5.- Definición de macros anidadas

El procesamiento de macro variables y expresiones en Clipper es totalmente dinámico permitiendo definiciones de macros anidadas. Por ejemplo, después de una asignación a una macro variable de otra macro variable, la macro variable original puede ser expandida con el resultado de la expansión de la segunda macro variable y la evaluación de su contenido. Por ejemplo:

```
cOne   := "&cTwo"
cTwo   := "cThree"
cThree := "Hello"
//
? &cOne             // Resultado "Hello"
```

8.- Arreglos

Un arreglo es una colección de items de datos relacionados que comparten el mismo nombre. Cada valor en un arreglo es referenciado como un elemento. Los elementos de arreglos pueden ser de cualquier tipo de datos; excepto memo, los cuales son un tipo de dato exclusivo. Por ejemplo, el primer elemento puede ser una cadena de caracteres, el segundo un valor numérico, el tercero un valor de fecha, y así sucesivamente. Los arreglos, interesantemente, pueden contener otros arreglos y bloques de códigos como elemento.

Los arreglos son referencias. Esto significa que una variable a la cual se le asigna un arreglo (o la cual es declarada como un arreglo) no tiene actualmente el contenido del arreglo. En cambio, ésta contiene una referencia al arreglo. Además, si una variable que contiene un arreglo es pasada como un argumento a un procedimiento o función usuario, una copia de la referencia es pasada. El propio arreglo nunca es duplicado.

TYPE() y VALTYPE() retornan "A" para un arreglo, el cual es distinguido como un tipo de dato. En Clipper 5.0, muchas de las funciones de arreglos existentes han sido cambiadas, para retornar la referencia al arreglo de modo que expresiones de arreglo complejas puedan ser creada. Las operaciones de arreglos válidas son discutidas en esta sección.

8.1.- Creando arreglos

PRIVATE y PUBLIC operan de la misma manera que en versiones anteriores de Clipper, con la excepción de que estas permiten la definición de arreglos multidimensionales. Además las sentecias LOCAL y STATIC han sido adicionadas al lenguaje para permitir explícitamente el alcance a los nombres de arreglos (ver Variables). Las sintaxis para la especificación de un arreglo con una de estas sentencias es cualquiera de las siguientes:

<identificador>[<nElementos1>, <nElementos2>, ...]

ó

<identificador>[<nElementos1>][<nElementos2>] ...

Diferente a las representaciones de otras sintaxis, los corchetes son parte de la definición de los arreglos y deben ser incluidos. A diferencia de la primera dimensión, <nElementos1>, todas las demás dimensiones son opcionales.

Con la excepción de la declaración de arreglos estáticos, cuyas dimensiones deben ser completamente especificadas en tiempo de compilación, los arreglos son dinámicos, permitiendo que el tamaño sea completamente determinado en tiempo de ejecución. Esto es, usted puede crear un arreglo cuyas dimensiones sean especificadas como expresiones:

```
i := 12 ; j := 4
LOCAL myArray[i, j]
```

La función ARRAY() puede ser usada también para crear un arreglo. Con esta función las dimensiones son especificadas como argumentos y el valor de retorno es un arreglo. Parecido a otras funciones, ARRAY() puede ser usada en expresiones sin argumentos.

Cada una de las sentencias de declaración de arreglo se traduce en dos partes: la declaración del nombre del arreglo con la subsecuente creación de un arreglo y la asignación de una referencia. Por ejemplo, PUBLIC myArray[12][4] podra ser igual a:

```
PUBLIC myArray := ARRAY(12,4)
```

Otras funciones que crean arreglos son DBSTRUCT() y DIRECTORY().

8.2.- Direccionando elementos de arreglos

Después que un arreglo es creado, sus elementos son accesados utilizando un índice entero, comúnmente referido como un subíndice. Para direccionar un elemento de arreglo, se pone el subíndice del elemento entre corchetes, siguiendo al nombre del arreglo. Por ejemplo, suponga que myArray es un arreglo unidimensional con 10 elementos. Para direccionar el primer elemento, usted puede usar:

```
myArray[1]
```

Note que la numeración del subíndice empieza con 1.

Para especificar más de un subíndice (por ejemplo, cuando usamos arreglos multidimensionales), usted puede encerrar cada subíndice por separado entre corchetes, o separar los subíndices con comas y encerrar la lista entre corchetes. Por ejemplo, si myArray es un arreglo bidimensional, las siguientes sentencias direccionan al elemento de la segunda columna y la décima fila:

```
myArray[10][2]
myArray[10,2]
```

Es ilegal direccionar un elemento que esta fuera del límite del arreglo. Intentar ésto resultará un error en tiempo de ejecución.

Cuando hacemos referencia a un elemento de arreglo usando un subíndice, usted está en ese momento aplicando el operador de subíndice ([]) en una expresión de arreglo. Una expresión de arreglo, por supuesto, es cualquier expresión que evalúe a un arreglo. Esto incluye llamadas a función, referencias a variables, operaciones subscritas, o cualquier otra expresión que evalúe a un arreglo. Por ejemplo, las siguientes son todas válidas:

```
{ "a", "b", "c" }[2]
x[2]
ARRAY(3)[2]
&(<macro-expresión>)[2]
(<expresión compleja>)[2]
```

8.3.- Asignando valores a elementos de arreglos

Cuando un arreglo es creado con una de las sentencias de declaración o con ARRAY(), cada elemento es creado con NIL hasta que sea inicializado con algún otro valor. La inicialización de arreglo es la asignación de valores a los elementos de éste acompañado del operador asignación (=) o de asignación en línea (:=).

En varios casos, es posible crear un arreglo e inicializarlo con una sola función. Por ejemplo, la función DBSTRUCT() crea un arreglo y lo inicializa con información de la estructura del fichero de base de datos. Similarmente DIRECTORY() inicializa el arreglo que crea con información del directorio del disco. El resultado de estas funciones puede ser asignado a variables, las cuales en su momento, serán referenciadas como arreglo.

Es además posible inicializar arreglos al mismo tiempo que son creados con las sentencias de declaración. Para llevar a cabo ésto no se define el arreglo con dimensiones en la sentencia de declaración. En cambio, solo se da el nombre de éste. Después del nombre del arreglo, se pone el operador de asignación en línea seguido por una expresión que retorne un arreglo. Por ejemplo, la siguiente sentencia crea un arreglo local y lo inicializa con información del directorio:

```
LOCAL myArray := DIRECTORY("*.*", "D")
```

La única limitación en esta implementación es con la sentencia STATIC la cual permite solo constantes y expresiones simples como asignación.

La función AFILL() es designada para inicializar los elementos de un arreglo ya existente, ya que usted no puede efectuar esto con una sentencia de simple asignación. El ejemplo a continuación le permite crear una variable numérica local cuyo valor es uno:

```
LOCAL myArray[30]
myArray := 1
```

mientras el siguiente ejemplo asigna un valor de uno a cada elemento en myArray:

```
STATIC myArray[30]
AFILL (myArray,1)
```

AFILL puede además ser usada para inicializar un rango de elementos. Por ejemplo, las siguientes líneas de código inicializan los primeros 10 elementos con uno, los segundos 10 elementos con 2 y los últimos 10 elementos con 3:

```
AFILL( myArray, 1,  1, 10 )
AFILL( myArray, 2, 11, 10 )
AFILL( myArray, 3, 21, 10 )
```

Los elementos de arreglos pueden además ser inicializados individualmente. Por ejemplo:

```
myArray[1] := 1, myArray[2] := 2,  myArray[3] := 3
myArray[4] := SQRT(4)
```

Los elementos de arreglo pueden ser inicializados usuando cualquier expresión que retorne un tipo de dato válido en Clipper, incluyendo bloques de códigos y otros arreglos. Una vez inicializados, los elementos de arreglos pueden ser usados como variables en cualquier lugar que sea apropiado para su tipo de datos.

8.4.- Arreglos multidimensionales

En Clipper 5.0, un arreglo multidimensional puede ser creado declarándolo con más de un parámetro de dimensión o asignando un arreglo a un elemento de otro arreglo ya existente.

Usted puede crear y mantener arreglos multidimensionales tradicionales que tienen un número fijo de elementos en cada dimensión. Por ejemplo, un arreglo bidimensional es regularmente usado para representar una tabla de filas y columnas. Al declarar un arreglo bidimensional con 10 filas y 2 columnas, usted puede usarlo así:

```
PUBLIC myArray [10][2]
```

Usualmente, los arreglos son creados de distintas maneras observando ciertas reglas. Por ejemplo, cada columna tiene el mismo tipo de información para cada fila en el arreglo (por ejemplo, la columna uno puede ser de cadena de caracteres, y la segunda columna, númerica). Ya que Clipper no impone ninguna regla de como salvar en un arreglo, este nivel de control debe implementarse programando.

El hecho que usted pueda asignar un arreglo a un elemento de arreglo ya existente le permite cambiar dinámicamente la estructura del mismo. Por ejemplo, después que un arreglo es creado, no existe algo que le prevenga de hacer cualquier cosa, parecido a esto:

```
myArray [1][2] := {1, 2, 3, 4, 5}
```

Esta asignación cambia a myArray significativamente porque uno de sus elementos es una referencia a un arreglo. En particular, una referencia a myArray [1][2][1] es ahora válida (esta retorna 1) que antes no existía. Referencias a myArray[1][1][1] y a myArray[2][1][1], en cambio, resultan errores de ejecución.

Esta característica de asignar una referencia de arreglo a un elemento de arreglo puede ser manejable en un lote de aplicaciones. La cuestión a recordar es que Ud como programador puede ejercer cualquier control que piense que sea necesario para almacenar y direccionar elementos de arreglo. Ud no puede hacer cualquier asunción alrededor de la estructura de un arreglo a menos que imponga una estructura.

8.5. - Arreglo de literales

Los arreglos de literales, llamados algunas veces arreglos de constantes, pueden ser creados en ejecución usando la siguiente sintaxis:

```
{ [ <exp> [, <exp> ... ]] }
```

Por ejemplo:

```
x := { 1,2,3 }
y := { "Hello", SQRT(x[2]), MyFun(x) }
```

Crear un arreglo semejante a éste, es igual a la declaración de un arreglo (por ejemplo, con PUBLIC o LOCAL) y entonces asignar los valores a cada elemento individualmente. Los arreglos de literales pueden ser usados en cualquier lugar, puede ser especificado, incluyendo como elemento, un arreglo literal. Por ejemplo para crear un arreglo literal bidimensional, Ud puede hacer lo siguiente:

```
aTrheeD := { {1,2,3}, {"a","b","c"}, {.t.,.t.,.f.} }
```

Si se expresa como una cadena de caracteres, un arreglo literal puede ser compilado con el macro-operador. Por ejemplo :

```
cArray := "{1,2,3}"
aNew := &cArray
? VALTYPE(aNew)              // retorna : A
```

Esto es muy provechoso ya que Ud no podría, en ningún caso, almacenar arreglos como campos de una base de datos, ni como valores de caracteres en un fichero texto. Note, sin embargo, la restricción del macro-compilador que sólo procesa cadenas de una longitud máxima de 254 caracteres.

8.6.- Arreglos como argumentos de funciones y valores de retorno.

Los arreglos pueden ser pasados como argumentos a procedimientos y funciones definidas por el usuario. Cuando un arreglo es especificado como un argumento a una rutina, éste es pasado por valor como una omisión, si la rutina es llamada usando las convenciones de llamada a función. Esto significa que una copia de la referencia es pasada a la variable parámetro que va a recibir el valor. También, la copia de la referencia pasa cualquier cambio hecho en el arreglo referenciado y son reflejados automáticamente en el original. Por ejemplo:

```
LOCAL myArray[10]       // todos elementos inicializados en NIL
AFILL (myArray,0)       // elementos inicializados en 0
MyFill(myArray)         // elementos incrementados en 1
//

? myArray[1]            // resultado es 1

FUNCTION MyFill (tempArray)
  FOR i = 1 TO LEN(tempArray)
    tempArray[i]++
  NEXT
  RETURN (NIL)
```

Cuando la función definida por el usuario MyFill es ejecutada, una copia de la referencia de MyArry es pasada a tempArray. Dentro de MyFill el valor del arreglo referenciado es incrementado en 1. Cuando la función retorna, la referencia que tuvo tempArray es descartada pero los cambios al arreglo referenciado aparecen después de la llamada.

Los arreglos pueden además ser retornados como valores desde funciones y procedimientos. Por ejemplo

```
myArray := MakeArray (10, "New Value")
//
FUNCTION MakeArray ( nElements, fillValue)
  LOCAL tempArray [nElements]
  AFILL(tempArray, fillValue)
  RETURN (tempArray)
```

8.7.- Recorriendo un arreglo

Hay dos formas de recorrer un arreglo en Clipper 5.0

8.7.1.- FOR . . . NEXT

El primero y más entendible es el uso del lazo FOR...NEXT. Esta construcción permite encontrar un elemento del arreglo cada vez usando la variable de control como el índice que define el elemento actual. En el siguiente ejemplo, a cada elemento en myArray le fue asignado el valor de su posición en el arreglo.

```
LOCAL myArray[10]
FOR i:= 1 TO 10
  myArray[i] := i
NEXT
```

Al recorrer un arreglo multidimensional, se podría anidar la construcción FOR...NEXT, un nivel para cada dimensión. El siguiente ejemplo muestra cada elemento en un arreglo bidimensional.

```
FUNCTION ArrDisp(myArray)
  LOCAL i,j
  //
  FOR i := 1 TO LEN(myArray)
    FOR j := 1 TO LEN (myArray[1])
       ? myArray[i][j]
    NEXT j
  NEXT i
RETURN (NIL)
```

Este método de recorrer un arreglo es totalmente tradicional y probablemente familiar si se tiene experiencia en lenguajes de alto nivel tales como el "C", Pascal o Basic.

8.7.2- AEVAL()

El otro método AEVAL(), requiere un dominio del bloque de código. Esta función evalúa un bloque de código para cada elemento de un arreglo, pasando el valor del elemento como un parámetro del bloque y retorna una referencia al arreglo. Por ejemplo la siguiente invocación de AEVAL() visualiza cada elemento de un arreglo unidimensional.

```
AEVAL(myArray, { |elemento| QOUT(elemento) } )
```

Para arreglos multidimensionales, se puede anidar funciones AEVAL() para recorrer cada dimensión. El siguiente ejemplo inicializa y visualiza un arreglo bidimensional.

```
LOCAL myArray[10][2], i:= 1
// llena cada elemento con el número de fila
AEVAL ( myArray, { |elemento| AFILL(elemento, i++) } )
// Muestra cada elemento
AEVAL(myArray,{|elemento|AEVAL(elemento,{|valor|QOUT(valor)})})
```

8.8.- Arreglos vacíos.

Un arreglo vacío es definido como un arreglo con 0 elementos y por lo tanto sin dimensiones. Para crear un arreglo vacío también se puede declarar un arreglo con 0 elementos o asignar un arreglo literal vacío a una variable. Por ejemplo las siguientes instrucciones son equivalentes:

```
LOCAL aEmpty[0]
LOCAL aEmpty := {}
```

Los arreglos vacíos pueden ser usados simpre que no se conozca en lo adelante la dimensión que pueda necesitarse. Después se le pueden adicionar elementos a un arreglo con la función AADD() or ASIZE().

Para verificar si un arreglo está vacío se usa la función EMPTY(). Por ejemplo:

```
FUNCTION ZeroArray (entity)
  IF VALTYPE(entity) = "A" .AND. EMPTY(entity)
     RETURN(.T.)
  ENDIF
  RETURN(.F.)
```

No se puede, por otra parte , verificar si un arreglo está vacío comparándolo con NIL. Esta comparación será siempre evaluada como (.F.)

8.9. - Determinando el tamaño de un arreglo.

El número de elementos de un arreglo puede ser determinado usando la función LEN(). Esta retorna el número de elementos contenidos en el arreglo o la posición del último elemento del arreglo. El siguiente fragmento de código lo demuestra:

```
LOCAL myArray[10][12]
? nArraySize := LEN (myArray)    // retorna 10
```

Note que este ejemplo retorna el número de elementos en la primera dimensión de myArray, recuerde que los arreglos bidimensionales son creados en Clipper por subarreglos anidados dentro del contenido de un arreglo. LEN() retorna el número de elementos especificados en el arreglo.

Para determinar el número de elementos en un sub- arreglo se puede usar la función LEN() al primer elemento del arreglo el cual retornará el número de elementos en la próxima dimensión anidada:

```
? nNestedSize := LEN(myArray[1])  // retorna 12
```

Nota : LEN() retorna 0 para un arreglo vacío

8.10.- Comparando arreglos.

El operador == es usado para comparar dos arreglos por equivalencia. Los arreglos citados son equivalentes si ellos son referenciados a un mismo arreglo (es decir, ellos apuntan a la misma localización en memoria). Por ejemplo:

```
LOCAL myArray := { 1, 2, 3 }
sameArray := myArray          // crea una nueva referencia a myArray
? sameArray == myArray        // resultado .T.
newArray  := ACLONE(myArray)  // crea un nuevo arreglo
? newArray  == myArray        // resultado .F.
```

Note que el operador == no es válido para comparar arreglos y que es un operador sencillo que no puede chequear si todos los elementos son iguales en dos arreglos distintos. Para llevar a cabo ésto se deben recorrer los arreglos y compararlos elemento a elemento. La siguiente función lleva a cabo ésto para arreglos unidimensionales:

```
FUNCTION ArrComp ( aOne, aTow )
  IF LEN(aOne) <> LEN(aTow)   // tamaños diferentes
    RETURN(.F.)
  ENDIF
  FOR i := 1 TO LEN(aOne)
    IF (aOne[i] <> aTow[i])  // al menos un elemento diferente
      RETURN(.F.)
    ENDIF
  NEXT
RETURN(.T.)                 // longitud y elementos iguales
```

8.11. - Cambiando el tamaño de un arreglo.

Existen dos funciones, ASIZE() y AADD() , que permiten cambiar el tamaño de un arreglo existente. ASIZE() especifica el arreglo a cambiar y el nuevo tamaño como argumentos de la función. Con esta función, Ud puede hacer un arreglo existente más largo o más corto. Por ejemplo:

```
LOCAL myArray[10]
ASIZE(myArray,25)    // incrementa myArray en 15 elementos
? LEN(myArray)       // retorna 25
ASIZE(myArray,5)     // decrementa myArray en 20 elementos
? LEN(myArray)       // retorna 5
```

Cuando Ud incrementa un arreglo (esto es hacerlo más largo) los nuevos elementos son adicionados al final del arreglo y se les asigna el valor NIL. Disminuir un arreglo (esto es hacerlo más pequeño) elimina elementos después de la nueva longitud de arreglo especificada. ASIZE() actualiza el arreglo y retorna una referencia a él.

AADD() agranda un arreglo en un elemento e inicializa el nuevo elemento con un valor en particular. El nuevo elemento es adicionado al final del arreglo. Por ejemplo:

```
LOCAL myArray [10]
AADD(myArray,500)            // adiciona un elemento
? LEN(myArray)               // retorna 11
? myArray[11]                // retorna 500
```

8.12. - Insertando y borrando elementos de arreglos

La función AINS() inserta un elemento dentro de un arreglo existente en una localización específica empujando los subsecuentes elementos hacia abajo una posición. El último elemento se pierde.

ADEL() borra un elemento en la posición especificada, y todos los elementos subsecuentes son movidos hacia arriba una posición. El último elemento se hace NIL.

Ambas funciones (ADEL() y AINS()) actualizan el arreglo original y retornan una referencia a éste. Diferente a AADD() y a ASIZE(), estas funciones no cambian la longitud del arreglo.

8.13.- Copiando elementos y duplicando arreglos

ACOPY() copia elementos desde un arreglo a otro. El arreglo destino debe existir anteriormente al invocar esta función. Con esta función, Ud. puede copiar todos los elementos o un rango particular de estos. Si el arreglo que usted quiere copiar contiene arreglos anidados, el arreglo destino contendrá solo la referencia a estos subarreglos, no la copia actual. ACOPY() retorna una referencia al arreglo destino.

Para arreglos multidimensionales o anidados, use ACLONE() si usted quiere duplicar el arreglo entero. Si el arreglo destino contiene un subarreglo, ACLONE() crea un subarreglo igualando y llenando éste con la copia de los valores del subarreglo original, de manera opuesta a la copia por referencia. Esta función crea un arreglo destino y retorna la referencia de éste.

El siguiente ejemplo ilustra la diferencia entre estas dos funciones:

```
LOCAL aOne[4], aTwo[14]
aOne[1] := { 1, 2, 3 }
ACOPY( aOne, aTwo )   // Copia de elementos desde aOne hacia aTwo
? aOne[1] == aTwo[1]     // Retorna .T.
aThree := ACLONE( aOne ) // Duplica aOne en este enteramente
? aOne[1] == aThree[1]   // Retorna .F.
```

El primer elemento en aOne es un arreglo. Después que ACOPY() es invocada para la copia de elementos hacia aTwo, los primeros elementos en esos arreglos son equivalentes (esto es, son referencias al mismo subarreglo). ACLONE() crea a aThree como un duplicado, y el chequeo de equivalencias fracasa porque los subarreglos iniciales, comparados no son iguales. No obstante, si aThree[1] y aOne[1] son comparados elemento a elemento bases, todos estos serán iguales.

8.14.- Ordenando un arreglo

La función ASORT() ordena un arreglo y retorna una referencia al nuevo arreglo ordenado. Con esta función usted puede organizar un arreglo entero o una porción especificada. Por ejemplo:

```
ASORT( myArray )
```

ordena el arreglo entero.

```
ASORT( myArray, 10, 5 )
```

ordena 10 elementos a partir del quinto dejando los demás en su sitio original. Una característica adicional ha sido adicionada a la función ASORT() permitiendo su propio criterio específico de organizar a través de un bloque de código. Por ejemplo, la siguiente sentencia organiza descendentemente un arreglo entero:

```
ASORT( myArray,,, { | x,y | x < y } )
```

8.15.- Buscando en un arreglo

ASCAN() es usada para buscar en un arreglo un valor particular el cual puede estar especificado como una expresión o como un bloque de códigos. Esta función puede operar en todo el arreglo o en un rango de elementos específico. Si es usada con una expresión simple de búsqueda, la función busca en el arreglo hasta encontrar un valor igualado y retorna el subíndice, o cero si el valor no fue encontrado. Por ejemplo:

```
LOCAL myArray := { "Tom", "Mary", "Sue", "Mary" }
? ASCAN( myArray, "Mary" ) // Result: 2
? ASCAN( myArray, "mary" ) // Result: 0
```

Si el criterio de búsqueda es especificado como un bloque de código, la función será de mucho más poder. En el ejemplo anterior ASCAN() no encuentra el nombre a buscar por estar en minúscula. El siguiente ejemplo usa un bloque de código para efectuar la búsqueda ignorando mayúsculas y minúsculas (case sensitive):

```
? ASCAN( myArray, { | x | UPPER( x ) == "MARY" } ) // Result: 2
```

Ya que la función permite un rango de subíndices específico a incluir en la búsqueda, ésta puede ser usada para resumir la búsqueda con el próximo elemento, como en el siguiente ejemplo:

```
LOCAL myArray := { "Tom", "Mary", "Sue", "Mary" }, nStart := 1
nEnd := LEN( myArray )              // Límite de la búsqueda
WHILE (nPos := ASCAN( myArray, "Mary", nStart )) > 0
    ? nPos, myArray[nPos]
    IF (nStart := ++nPos) > nEnd    // Chequea límite
      EXIT
    ENDIF
END
```

9.- Bloques de código

Los bloques de código proveen recursos de exportación en pequeñ as piezas de código ejecutable de programa desde un sitio en un sistema a otro. Usted puede pensar en ellos como si fueran funciones asignables no nombradas. Ellas son asignables porque, excepto cuando se ejecutan, Clipper las trata como valor. Esto es, ellas pueden ser salvadas en variables, pasadas como argumentos, etc.

Los bloques de código poseen una fuerte semejanza con las macro, pero con un significado diferente. Las macro son cadenas de caracteres que son compiladas al vuelo (on the fly) en tiempo de ejecución y ejecutadas inmediatamente. Los bloques de código, por el contrario, son compilados junto al resto del programa en tiempo de compilación. Por esta razón los bloques de código son más eficiente que las macro, a la vez que ofrecen similar flexibilidad.

La diferencia entre un bloque de código y las macros llega a ser especialmente importante cuando usamos variables declaradas. La declaración de variables actúa en tiempo de compilación y, no tienen efecto dentro de la ejecución de las macros. Puesto que los bloques de código son compilados en tiempo de compilación, la declaración puede ser usada para controlar el acceso a las variables en los bloques de código. Las variables que requieren declaración en tiempo de compilación (variables locales, estáticas, y parámetros declarados) no son visibles dentro de la ejecución de las macros, por el contrario ellas pueden ser accesadas libremente en los bloques.

9.1.- Definiendo un bloque de código

La sintaxis de un bloque de código es la siguiente:

{ | [<lista de argumentos>] | <lista de expresiones> }

Tanto la <lista de argumentos> como la <lista de expresiones> son separadas por comas (,). Las expresiones deben ser expresiones Clipper válidas -ellas no pueden contener comandos o sentencias como estructuras de control y declaraciones.

Nota: La barra vertical que delimita la lista de argumentos de un bloque de código debe estar presente -aún si no existen argumentos- para distinguir un bloque de código de un arreglo literal.

Algunos ejemplos de bloques de código son los que siguen:

```
{ || "Esto es una cadena" }
{ | p | p + 1 }
{ | x, y | SQRT( x ) + SQRT( y ) }
{ | a, b, c | myFunc( a ), myFunc( b ), myFunc ( c ) }
```

9.2.- Operaciones

La siguiente tabla muestra una sinópsis de todas las operaciones válidas en lenguaje Clipper para los bloques de código. Estas operaciones actúan en uno o más bloques de código para producir un resultado. El tipo del resultado no es necesariamente un bloque de código.

Tabla 1-18: Operaciones de los bloques de código

Operación	Descripción
=	Asignación
:=	Asignación en línea
AEVAL()	Evalúa un bloque de código para elementos de un arreglo
DBEVAL()	Evalúa un bloque de código para artículos de una dBF
EVAL()	Evalúa un bloque

9.3.- Ejecutando un bloque de código

La función EVAL() es usada para ejecutar un bloque de código. EVAL() toma un bloque como su primer argumento, seguido por una lista de parámetros. La función entonces ejecuta el bloque de código, las expresiones son evaluadas de izquierda a derecha. El valor retornado por el bloque de código es el resultado de la última (o la única) expresión en la lista. Por ejemplo:

```
bBlock := { | nValue | nValue + 1 }
? EVAL( bBlock, 1 )  // Result: 2
```

Las funciones AEVAL() y DBEVAL() también pueden ejecutar bloques de código. Estas son funciones iterativas que procesan un arreglo o un fichero base de datos respectivamente, ejecutando el bloque de código para cada elemento o artículo.

9.4.- Alcance de las variables dentro de un bloque de código

Cuando un bloque de código es ejecutado, las referencias a variables locales y estáticas (también argumentos) tienen alcance hasta el procedimiento o función en el cual el bloque de código fue definido. Debido a que un bloque de código puede ser pasado como parámetro a cualquier programa, usted puede exportar variables estáticas y locales. Por ejemplo:

```
FUNCTION One()
  LOCAL myVar
  myVar := 10
  bBlock := { | number | number + myVar }
  //
  NextFunc( bBlock )
  //
  RETURN NIL

FUNCTION NextFunc( bBlock )
  RETURN ( EVAL( bBlock, 200 ) )
```

Cuando bBlock es evaluado en NextFunc(), myVar, la cual es local en la función One() llega a ser visible aún no habiendo sido pasado directamente como parámetro.

Note que si la variable especificada es una variable pública o privada no es exportada de la misma forma definida en la rutina. En este caso la visibilidad actual alcanza la variable usada.

9.5.- Usando bloques de código

El bloque es un tipo de dato especial que contiene un código de programa ejecutable. Un bloque puede ser asignado a una variable o pasado como parámetro exactamente del mismo modo que otros tipos de datos.

Los bloques de código son particularmente convenientes cuando creamos funciones cajas negras (black-box) las cuales pueden operar sin el conocimiento del tipo de información que ha sido pasado a ellas. Considere el siguiente caso:

```
FUNCTION SayData( dataPassed )
  // Muestra el contenido de dataPased como una cadena
  DO CASE
    CASE VALTYPE(dataPassed) = "C"            // Caracter
        @10,10 SAY dataPassed
    CASE VALTYPE(dataPassed) = "N"            // Numerico
        @10,10 SAY LTRIM( STR( dataPassed ) )
    CASE VALTYPE(dataPassed) = "L"            // Logica
        @10,10 SAY IF( dataPassed, "True", "False" )
  ENDCASE
  RETURN ( .T. )
```

En este ejemplo un error puede ocurrir si el dato pasado a SayData() es de otro tipo que los especificados. Si, por ejemplo, una fecha necesita ser mostrada, SayData() debe ser modificada.

En el ejemplo abajo, el código a ejecutar es salvado en un bloque:

```
charCode := { | data | data }
numCode  := { | data | LTRIM( STR( data ) ) }
logCode  := { | data | IF( data, "True", "False" ) }
dateCode := { | data | DTOC( data ) }
```

SayData() ahora puede ser definida como una rutina caja negra la cual puede actuar con cualquier tipo de dato. La decisión de cual bloque es pasado a SayData() debe ser hecha en el programa que llama:

```
FUNCTION SayData( dataPassed, codeBlock )
  // Muestra el contenido de dataPased como una cadena
  @10, 10 SAY EVAL( codeBlock, dataPassed )
  RETURN ( .T. )
```

Los bloques de código pueden además ser usados para pasar una expresión a una función o procedimiento que usted no quiere que sea ejecutada hasta un tiempo más tarde. Un buen ejemplo de esto puede encontrarse en STD.CH con la definición de los comandos SET KEY y SET FUNCTION. Ambos comandos traducen dentro de una llamada la misma función aunque la operación no es efectuada de la misma forma en ambos casos: SET KEY asigna el nombre de un procedimiento o función definida por el usuario que será llamada cuando una tecla específica es presionada, mientras SET FUNCTION asigna una cadena a insertar dentro del buffer de teclado.

9.6.- Almacenando y compilando bloques de código en tiempo de ejecución

Aunque los bloques de código no pueden ser salvados directamente en una base de datos o fichero .MEM, ellos pueden ser salvados como cadenas de caracteres. Cuando usted quiere usar la definición de un bloque de código almacenado como una cadena de caracteres en un fichero de una base de datos en un programa, compile el campo usando el macro operador y entonces asigne el bloque de código resultante a una variable. Por ejemplo:

```
USE DataDict NEW
//
```

```
// Crea un bloque como cadena y asigna ésta a una variable campo
APPEND BLANK
REPLACE BlockField WITH "{ || Validate() }"
 .
 .
 .
// Después compila y ejecuta el bloque

bSomeBlock := &(BlockField)
EVAL(bSomeBlock)
```

En vez de almacenar el bloque de código compilado y entonces evaluarlo, usted puede compilar y evaluar en un solo paso con EVAL(&(BlockField)). Existe, sin embargo, una ventaja en rapidez al almacenar el bloque de código compilado en una variable ya que el compilador tuvo que dar un solo paso para ejecutarlo una vez. Una vez compilado, la variable puede ser pasada a lo largo del sistema y evaluada a la misma velocidad que otros códigos Clipper compilados.

Nota: El macro operador es apropiado solo para usar con expresiones de caracter. Por consiguiente, usted no puede compilar un bloque de código directamente con el macro operador.

10.- Objetos y mensajes

Clipper 5.0 ofrece una implementación limitada del tipo de dato especial denominado objeto. La noción de un objeto es basada en la tecnología de programación orientada a objeto.

Nota: Clipper 5.0 no es un lenguaje orientado a objeto. La implementación de objetos en Clipper 5.0 representa sólo un pequeño subconjunto del paradigma orientado a objeto.

Los objetos en Clipper 5.0 son simplemente valores de datos complejos los cuales tienen una estructura pre-definida y un conjunto de conductas. Un pequeño número de tipos de objetos pre-definidos -denominados clases- son suministradas con el sistema. Estas clases son designadas para soportar operaciones particulares en Clipper 5.0. La creación de nuevas clases o subclases de clases existentes, no está habilitada.

10.1.- Clases

En Clipper 5.0, existen diferentes tipos de objetos. Un tipo de objeto es conocido formalmente como clase (class). La información contenida en un objeto y las operaciones que pueden ser aplicadas a éste varían en dependencia de la clase de objeto. Para cada clase habilitada existe una función especial llamada "create function". Esta facilita la creación de nuevos objetos asociados a una clase.

Las clases habilitadas y los nombres de las "create function" son discutidas en el capítulo de Clases Standard de este libro.

10.2.- Instancias

Con una llamada a la " create function" para una clase particular, un nuevo objeto es creado. El objeto puede tener los atributos y conducta específica en la descripción de las clases. El nuevo objeto es formalmente conocido como una instancia de una clase.

Semejante a los arreglos en Clipper, los objetos son referencialmente manipulados. Esto significa que una variable de programa no puede actualmente contener un objeto. Permite, sin embargo, contener una referencia a un objeto. La variable es entonces mencionada por referencia al objeto y las operaciones pueden ser ejecutadas en el objeto por la aplicación del operador de envio (:) a la variable. Las referencias a objetos pueden ser libremente asignadas, pasadas como parámetros y retornadas desde funciones. Esto es posible por dos o más variables que contienen una referencia al mismo objeto (por ejemplo, si una variable que contiene una referencia a objeto es asignada a otra variable). En este caso, ambas variables pueden referir al objeto y las operaciones pueden ser ejecutadas en el objeto por la aplicación del operador de envio en cualquiera de las dos variables.

Los objetos continúan su existencia tanto como la referencia a ellos esté activa en algún sitio del sistema. Cuando todas las referencias a un objeto son eliminadas, el espacio ocupado por éste es automáticamente reclamado por el sistema.

Cuando un nuevo objeto es creado, la "create function" retorna una referencia al nuevo objeto. Esta referencia entonces puede ser asignada a una variable y el objeto puede ser accesado a través de ésta.

10.3.- Variables instancias

Un objeto contiene dentro de él toda la información necesaria para ejecutar las operaciones especificadas por su clase. Esta información es almacenada dentro del objeto en unas localizaciones de almacenamiento especiales denominadas variables instancias.

Cuando un nuevo objeto es creado este recibe de si mismo un conjunto dedicado de sus variables instancias. Las nuevas variables instancias son automáticamente asignadas con valores iniciales.

Las variables instancias son invisibles para el programa. No obstante, algunas variables instancias son accesibles. Estas son conocidas como variables instancias exportadas. Estas pueden ser inspeccionadas y -en algunos casos- asignadas usando el operador de envio.

10.4.- Enviando mensajes

Cada clase define un conjunto de operaciones que pueden ser ejecutadas en objetos de estas clases. Estas operaciones son ejecutadas por "envio de mensaje" al objeto usando el operador de envio (:).

La sintaxis para enviar mensajes a un objeto es la siguiente:

<objeto>:<selector> [(<lista de argumentos>)]

Por ejemplo:

myBrowse:pageUp()

En este ejemplo, myBrowse es el nombre de una variable que contiene una referencia a objeto. pageUp() es un selector. El selector especifica la operación que será ejecutada. Las operaciones habilitadas (y sus correspondientes selectores) varían en dependencia de la clase de objeto (ver en la descripción de clases la lista de operaciones habilitadas y selectores).

Los paréntesis son opcionales si los parámetros no son suministrados en el envio de mensajes. Por convención, de todos modos los paréntesis son usados para distinguir el envío de mensajes sin parámetros de un acceso a una variable instancia exportada.

Cuando un mensaje es enviado a un objeto, el sistema examina el selector. Si el objeto es de una clase que define una operación para ese selector, el sistema automáticamente invoca un "método" para ejecutar la operación en el objeto especificado. Si la clase no define un método para el selector especificado, ocurre un error de ejecución.

La ejecución de un envío de mensaje produce un valor de retorno muy semejante a una llamada a función. El valor retornado varía en dependencia de la operación ejecutada.

10.5.- Accesando a variables instancias exportadas

Algunas clases permiten el acceso a las variables instancias dentro de objetos de esas clases. Estas variables son llamadas variables instancias exportadas. La definición para una clase, especifica cuales variables instancias existentes son exportadas. Algunas variables instancias exportadas son mencionadas como asignables. Una variable instancia asignable puede ser modificada usando una variante especial del operador de envío de mensaje.

Las variables instancias exportadas son accesadas usando una variante de la sintaxis normal del envío de mensaje. La sintaxis para actualizar el valor de una variable instancia es:

<objeto>:<selector>

La sintaxis para la asignación de un nuevo valor a una variable instancia exportable asignable es:

<objeto>:<selector> := <nuevo valor>

Por ejemplo:

```
row           := myBrowse:cursRow
myBrowse:cargo := cargoValue
```

Note que el selector en una variable exportada accesa especificando cual variable es accesada en vez de especificar una operación para ser ejecutada en el objeto.

Cuando un nuevo valor es asignado a una variable instancia, la operación retorna el valor asignado. En otro contexto éste es compatible con el operador de asignación.

11.- El sistema de bases de datos

El sistema de bases de datos Clipper posee areas de trabajo ficticias que son usadas para manipular bases de datos y otros ficheros relacionados, así como otras operaciones para manipular estos ficheros. Un fichero de base de datos es una colección de información relacionada la cual es almacenada en forma de una tabla. La designación de la tabla, conocida como la estructura del fichero, es también almacenada en el fichero de base de datos.

La estructura del fichero es definida y adicionada al fichero de base de datos cuando éste es creado. Esta consiste de una o más definiciones de campos, describiéndose el nombre, tamaño, ancho y el tipo de datos para cada columna en la tabla. Las filas de la tabla, o artículos, son adicionados al fichero usando las operaciones de adjuntar conocidas e imponiendo la estructura del fichero sobre los campos según el campo básico. Los artículos son adicionados al fichero de base de datos en el orden físico que éste es usado, por defecto, cuando se produce el acceso al mismo. Un fichero índice es un fichero subordinado que es creado separadamente del fichero de base de datos. Este le permite a usted definir y mantener un ordenamiento lógico para su fichero de base de datos sin afectar el orden físico del mismo.

Varios ficheros de bases de datos que tienen estructuras y datos relacionados pueden ser asociados a todo lo largo con sus ficheros índices, usando el comando SET RELATION. SET RELATION permite establecer relaciones entre varios ficheros y operar sobre ellos como una única entidad conocida como una base de datos o "vista".

El utilitario de bases de datos, DBU.EXE, es proporcionado como parte del paquete Clipper 5.0 para permitirle ejecutar operaciones de bases de datos e índices usando un sistema de menúes. Esta es una exelente herramienta para la designación de estructuras de ficheros de bases de datos para sus aplicaciones y para comprobar los datos. Refiérase al capítulo Database Utility en el libro Programming and Utilities para más información relativa a DBU.

Esta sección describe las múltiples facetas del sistema de bases de datos y proporciona una breve introducción de todas las operaciones de bases de datos e índices.

11.1.- Areas de trabajo

El sistema de bases de datos Clipper define 255 áreas de trabajo -250 de ellas estan disponibles para su uso, y las 5 restantes son reservadas para uso interno. Un área de trabajo es esencialmente un área en memoria en la cual un fichero de base de datos puede ser manipulado con un fichero MEMO opcional y hasta 15 ficheros índices. Un área de trabajo es descrita como "ocupada" o "desocupada" en dependencia de si tiene o no un fichero abierto. Al iniciarce una aplicación todas las áreas estan desocupadas, y uno (1) es el actual, o activo, número de área de trabajo.

11.1.1.- Accesando área de trabajo

Al abrir un fichero de base de datos usted debe primero acceder al área de trabajo que quiere usar, y la forma más común de hacer esto es con el comando SELECT. SELECT es designado específicamente para movernos entre las áreas de trabajo cambiando del área de trabajo actual hacia una que usted especifique.

Las reglas más generales para el uso del SELECT son las siguientes: Para acceder a un área de trabajo específica no ocupada, especifique SELECT con un número entre 1 y 250; para acceder a la próxima area de trabajo disponible, use SELECT 0; al desear trabajar con un área activa y un fichero de base de datos esta en uso (es decir, el área de trabajo esta ocupada), especificar SELECT con el nombre del área, es más cómodo que especificar el área por el número.

En Clipper 5.0, el comando USE soporta una cláusula NEW que selecciona la próxima área desocupada antes de ejecutar la operación de apertura. Esta característica deja obsoleto al comando SELECT 0; cualquier código de la forma:

SELECT 0
USE <xcDatabase>

puede ser reemplazado por:

USE <xcDatabase> NEW

Después de un USE el área de trabajo seleccionada permanece activa.

Si la claúsula ALIAS no es especificada, como en el ejemplo anterior, al área de trabajo en la cual el fichero de base de datos es abierto le es asignado como nombre alias el mismo que el del fichero base de datos. Este nombre es usado por SELECT para acceder a dicha área.

Las expresiones que utilizan ALIAS son usadas para acceder temporalmente a un área de trabajo con el propósito de la evaluación de una expresión. Para formar una expresión con ALIAS, enciérrela entre paréntesis y prefíjela con el nombre de ALIAS deseado y el operador (por ejemplo, <idAlias>->(<exp>)). El área de trabajo va a ser seleccionada, la expresión evaluada y finalmente el área de trabajo original es restaurada.

Las funciones simples de bases de datos y las más complicadas expresiones pueden ser evaluadas para un área no seleccionada en esta forma. Por ejemplo: * Cust->(EOF()) evalúa el estado del fin de fichero en el área de trabajo Cust.

*Cust->(HEADER() + (LASTREC() * RECSIZE())) calcula el tamaño del fichero en el área de trabajo Cust.

* Cust->(total + Cust->Amount) adiciona el valor del campo Amount en el área de trabajo Cust a la variable "total".

Advierta que en el final del ejemplo anterior, Amount está referenciada explícitamente con su nombre de área sin paréntesis dentro de la expresión ya referenciada por un ALIAS. Esto es una referencia explícita a campos para evitar cualquier ambiguedad que pueda existir entre campos y nombres de variables e ilustrar el punto siguiente.

Nota: El uso de una expresión con ALIAS no evita la asunción del compilador acerca de las variables dentro de la expresión, visto que, para hacer referencia a un nombre de campo debe usarse el ALIAS.

En otras palabras, el compilador no asume que todo dentro de una expresión refiere a un campo por el solo hecho de que la misma sea precedida por un ALIAS. En cambio, las reglas standard de precedencia son usadas para resolver ambiguedades cuando la expresión es evaluada en tiempo de ejecución. Los campos que estan explícitamente referenciados con un ALIAS, por otra parte, son referencias ambiguas que son resueltas en tiempo de compilación.

Para ilustrar, suponga que usted tiene el código que sigue, que usa una variable local Amount y un campo llamado Amount en el área de trabajo Cust:

```
LOCAL  total := 0, amount := 0
total := Cust->(total + amount)
```

Esta expresión usada con ALIAS accede a la variable local llamada Amount porque cuando la expresión es evaluada, la variable es visible y toma la precedencia sobre el campo del mismo nombre. Cambiando el código hacia:

```
LOCAL  total := 0, amount := 0
total := Cust->(total + Cust->amount)
```

accede al campo Amount antes que a la variable local porque la referencia explícita al campo a través de su ALIAS pasa por encima a la sentencia de declaración LOCAL.

Como una buena práctica de programación, usted deberá hacer todas las referencias a identificadores explícitas y evitar los conflictos por nombramientos siempre que sea posible. De lo contrario usted dependerá del compilador para resolver las ambiguedades si ellas ocurrieran, en cuyo caso puede ser que usted no este de acuerdo con la decisión del mismo.

11.1.2.- Atributos del área de trabajo

Un área de trabajo define un significado para un grupo de atributos lógicos que pertenecen a un fichero de base de datos. Estos son llamados atributos del área de trabajo, ellos aparecen en la tabla siguiente:

Tabla 1-19: Atributos del área de trabajo

Funciones/Comandos	Atributos
SET DELETED	Filtrador de artículos borrados
SET FILTER	Filtrador de artículos lógico
ALIAS()	Nombre del "alias" de una B.Datos
BOF()	Bandera de inicio de fichero
DBFILTER()	Condición SET FILTER
DBRELATION, DBRSELECT()	Información SET RELATION
EOF()	Bandera de fin de fichero
FCOUNT()	Cantidad de campos
FOUND()	Bandera de búsqueda
FLOCK(),RLOCK()	Bandera de estado del lock
INDEXKEY()	Expresión de la llave índice
INDEXORD()	Valor del SET ORDER

LOCATE, CONTINUE	Condición del LOCATE
RECCOUNT() | LASTREC()	Cantidad de artículos
RECNO()	Número del artículo
SELECT()	Selección de área

11.2.- Ficheros de bases de datos

El tipo principal de fichero es el fichero de base de datos el cual tiene extensión implícita (.dbf). Un fichero de base de datos, regularmente llamado tabla, consiste de un artículo de encabezado de longitud variable que define la estructura del fichero en términos de sus definiciones de campos, y cero o más artículos de longitud fija que contienen los datos actuales para campos que no son MEMO y la información del apuntador a los campos MEMO. Cada artículo tiene un byte adicional para la bandera de estado de borrado. El formato de bases de datos Clipper es compatible con el dBASE III PLUS. Los ficheros de bases de datos pueden dar idea de una tabla en la que cada campo en la estructura del fichero representa una columna de la tabla y cada artículo una fila.

11.2.1.- Ficheros MEMO

Si una estructura de un fichero base de datos tiene definiciones de uno o más campos MEMO, existe un fichero MEMO asociado con el fichero de base de datos. Los ficheros MEMO tienen una extensión implícita (.dbt) y tienen el mismo nombre que el fichero base de datos asociado. Por defecto, los ficheros bases de datos y MEMO son creados y mantenidos en el mismo subdirectorio.

El dato para todos los campos MEMO es almacenado en el mismo fichero MEMO, sin importar cuantos campos MEMO estan definidos en la estructura del fichero de base de datos. Cada campo MEMO en el fichero de base de datos contiene un apuntador dentro del fichero MEMO, permitiéndole al sistema de bases de datos localizar rápidamente el dato asociado. Los ficheros MEMO son automáticamente abiertos y mantenidos en la misma área de trabajo que el fichero de base de datos.

11.2.2.- Atributos de los ficheros de bases de datos

Al igual que las áreas de trabajo, los ficheros de bases de datos tienen asociados atributos que aparecen listados en la tabla siguiente.

Tabla 1-20: Atributos de los ficheros de bases de datos

Función	Atributo
DELETED()	Bandera de estado de artículo borrado
FIELD()	Nombre de campo
HEADER()	Tamaño del encabezamiento en bytes
LUPDATE()	Fecha de la última actualización
RECSIZE()	Tamaño del artículo en bytes

11.2.3.- Operaciones de bases de datos

Existen varias operaciones que pueden ser ejecutadas sobre un fichero de base de datos abierto las cuales son conocidas como operaciones de bases de datos. La tabla a continuación categoriza todas las operaciones de bases de datos. Note que todas estas operaciones requieren que un fichero de base de datos este abierto en el área seleccionada actualmente.

Tabla 1-21: Operaciones de bases de datos

Categoria	Comando/Función
Adiciona	APPEND BLANK, APPEND FROM, BROWSE()
Cerrar	CLOSE, USE
Calcular	AVERAGE, COUNT, SUM
Crear	COPY STRUCTURE, COPY STRUCTURE EXTENDED, COPY TO, CREATE, CREATE FROM, JOIN, SORT, TOTAL
Borrar	DELETE, PACK, RECALL, ZAP
Mostrar	DISPLAY, LABEL FORM, LIST, REPORT FORM, BROWSE(), DBEDIT()
Filtrar	INDEX ... UNIQUE, SET DELETE, SET FILTER
Información	AFIELDS()
Iteración	DBEVAL()
Lock	SET EXCLUSIVE, UNLOCK(), FLOCK(), RLOCK()
Recorrido	CONTINUE, FIND, GO, LOCATE, SEEK, SKIP
Abrir	USE
Ordenar	INDEX, REINDEX, SET INDEX, SET ORDER, SORT
Relación	SET RELATION
Actualizar	@ GET <idField>/READ, REPLACE, UPDATE, BROWSE(), DBEDIT()

11.2.4.- Alcance de los artículos

Muchas de las operaciones de bases de datos mencionadas anteriormente pueden procesar subconjuntos de artículos dentro de un área usando un alcance y claúsulas condicionales. Para cualquier comando de bases de datos que permita un alcance <scope> como parte de su sintáxis, la sintáxis del <scope> es como sigue:

[ALL | NEXT <nRecords> | RECORD <nRecord> | REST]

* ALL procesa todos los artículos

* NEXT procesa el artículo actual y el número de artículos especificado.

* RECORD procesa el artículo especificado

* REST procesa todos los artículos desde el actual hasta el fin de fichero

Los comandos sin un alcance especificado, toman como implícito al artículo actual (NEXT 1) o todos (ALL) los artículos, dependiendo del comando. Por ejemplo, DELETE y REPLACE son típicamente usados para procesar el artículo actual y por lo tanto el implícito es (NEXT 1). Otros comandos, tales como REPORT y LABEL FORM, son típicamente usados para procesar un conjunto de artículos y por lo tanto el ímplito es (ALL) todos los artículos. La especificación de un alcance cambia este valor implícito para indicar cuántos artículos procesar y dónde comenzar.

El conjunto de artículos procesados puede además ser restringido usando una claúsula condicional que especifica un subconjunto de artículos sobre la base de una condición lógica. Las dos claúsulas condicionales son FOR y WHILE.

La claúsula FOR define una condición que cada artículo dentro del alcance debe satisfacer en el orden que son procesados. Si otro <scope> no es especificado, FOR cambia el alcance implícito a todos (ALL) los artículos.

Una claúsula WHILE define otra condición que cada artículo procesado debe satisfacer; tan pronto como un artículo encontrado cause que la condición no se cumpla, el comando termina. Si otro alcance no es especificado WHILE cambia el alcance implícito hacia (REST).

En la especificación de un alcance, una claúsula FOR y WHILE dentro de la misma sintáxis del comando pueden suscitar preguntas relacionadas con el orden en que las claúsulas son procesadas. Entonces, la claúsula WHILE es evaluada y si la condición no es satisfecha el proceso termina. Si la condición del WHILE es satisfecha, la claúsula FOR es evaluada. Si la condición del FOR es también encontrada, el artículo es procesado; de lo contrario esto no ocurre. En cualquier caso el apuntador al artículo es movido al próximo dentro del alcance hasta que éste se agote.

11.2.5.- Procesamiento primitivo de artículos DBEVAL()

La operación de iteración, DBEVAL(), es nueva en Clipper 5.0 y por lo tanto, merece una explicación amplia. Esta función es original de las bases de datos y es usada en STD.CH para definir comandos de bases de datos que permiten alcanzar artículos lo cual es llamado por lo regular procesamiento primitivo de artículos.

DBEVAL() evalúa un código de bloque para cada artículo comparando el alcance especificado y las condiciones. Se usa para crear comandos usuarios de procesamiento de bases de datos. Esta función aparece en el capítulo de funciones standard que tiene más información y ejemplos.

11.3.- Ficheros índices

Los ficheros de bases de datos son mantenidos y procesados en el orden físico particular o sea, en el mismo orden en el cual los artículos fueron introducidos al fichero. Cada artículo tiene un número sequencial comenzando con 1 (uno). El número del artículo actual puede ser accesado con la función RECNO(). El indexado se hace a partir de un ordenamiento lógico del fichero de base de dato de acuerdo al tipo de dato que éste posee. En Clipper el indexado se lleva a cabo usando ficheros índices que usted crea y abre en un área de trabajo con un fichero de base de datos.

Por defecto, Clipper usa su propio método para la creación y mantenimiento de ficheros índices los cuales no son compatibles con otros dialectos. Este método crea ficheros índices con una extensión implícita (.ntx).

Un método alternativo es proporcionado en la forma de manipular bases de datos para soportar ficheros índices (.ndx) pertenecientes al dBASE III PLUS. Si usted usa éste manipulador de bases de datos con su aplicación, entonces es usado el método alternativo de indexado y los ficheros índices son creados con la extensión implícita (.ndx).

11.3.1.- Creando

Los ficheros índices son creados para el fichero de base de datos activo en el área de trabajo actual usando el comando INDEX en el cual usted especifica la expresión llave. La longitud máxima de la expresión llave es de 250 carácteres y puede ser de tipo carácter, numérico o fecha.

El valor de la llave de índice para cada artículo en la base de datos es determinado usando la expresión llave del índice. Esa expresión es guardada en el fichero índice de modo que el fichero de base de datos puede ser procesado en el orden de acuerdo a la llave de indexado, cuyo valor puede ser buscado rápidamente.

La cláusula UNIQUE del comando INDEX y la bandera SET UNIQUE ON crean un fichero índice en el cual la primera ocurrencia de cada valor de llave es mantenido en el fichero índice. El atributo UNIQUE es guardado en el fichero índice.

11.3.2.- Abriendo

Se pueden crear varios ficheros índices para una sola base de datos de modo que los datos pueden ser accesados de diferentes modos. Una vez que el fichero índice es creado, usted puede abrirlo asociado a una base de datos en

el orden de uso. Hasta 15 ficheros índices pueden ser abiertos al mismo tiempo con la clausula INDEX del comando USE o con SET INDEX.

Semejante a todas las operaciones de abrir ficheros de base de datos en Clipper, USE y SET INDEX primero buscan en el drive establecido por SET DEFAULT y en el directorio para el fichero abierto. Si el fichero no es encontrado, es buscado en la lista de SET PATH en el orden especificado. Solo después de haber buscado por todos los caminos se realizará la operación de apertura o el fracaso.

11.3.3.- Ordenando

Con USE y SET INDEX usted indica la lista de ficheros índices que desea abrir, siendo el primero el que controla el índice. El índice controlador define el orden en el cual el fichero de base de datos será procesado y será la única lista de índices por la que se buscará.

El comando SET ORDER permite cambiar la lista controladora por la que se encuentra en la posición especificada en la lista de ficheros índices. SET ORDER TO 0 es usada para especificar el orden físico de la base de datos como si no se usará lista índice.

11.3.4.- Buscando

SEEK y FIND son usadas para localizar un artículo a partir del valor de la llave índice en el fichero índice controlador. SEEK es el comando preferido para la búsqueda; FIND es un comando obsoleto que se ofertó solo por un pro- blema de compatibilidad.

Clipper usa uno de los dos métodos de búsqueda dependiendo de una bandera global fijada, SET SOFTSEEK. Si SET SOFTSEEK está en OFF (por omisión), una búsqueda infructuosa mueve el puntero de artículos hasta el fin de la base de datos, fijando la bandera de EOF() como verdadero (.T.). Si SET SOFTSEEK está en ON, la búsqueda infructuosa mueve el puntero hasta el próximo valor más cercano de la llave índice.

La función FOUND() es usada para chequear la ocurrencia o no de fallo en la operación de búsqueda.

11.3.5.- Actualizando

Mientras un fichero índice está abierto, cualquier cambio hecho en el fichero de base de datos es reflejado automáticamente en el fichero índice. Esto es así para todas las listas índices abiertas y no solo para la lista índice controladora. Los cambios a una base de datos son hechos usando cualquier operación de actualización de bases de datos mencionadas anteriormente en ésta sección.

Además, las operaciones de bases de datos que mueven el puntero de artículo causan que el fichero índice sea actualizado antes de que se mueva el puntero del artículo si el valor de la llave índice a sido cambiado. Una operación COMMIT, o el cierre de un fichero índice puede también actualizar la lista índice.

El comando REINDEX vuelve a crear todos los ficheros índices abiertos en el área de trabajo actual organizando cada artículo del fichero base de datos.

11.3.6.- Cerrando

Los ficheros índices son cerrados utilizando SET INDEX TO sin parámetros o con CLOSE INDEX. Cualquier operación de cierre de bases de datos cierra todos los ficheros índices abiertos en la misma área de trabajo.

12.- Sistema de Entrada/Salida.

El lenguaje Clipper provee un sistema completo de Entrada/Salida que permite la entrada de datos y la muestra de la información en pantalla y en la impresora. El sistema incluye una validación de datos propia que es extensiva a través del uso de las funciones definidas por el usuario. Se incluyen facilidades que permiten crear simples reportes por columnas y MAILING LABELS, y con los comandos y funciones de bibliotecas ofertados, usted puede crear mecanismos de reporte que se ajusten a sus necesidades. Esta sección lo introduce al sistema de entrada/salida de Clipper dándole una información básica del control de la impresora y la consola.

12.1.- Operaciones de la consola

Los comandos y funciones de salida por display sin referencia a la posición de la fila y columna son llamadas operaciones de consola. Usadas conjuntamente con el SET ALTERNATE y SET PRINTER, estas operaciones son capaces de enviar simultáneamente la salida a la pantalla, un fichero texto o la impresora. No obstante, la mayoría de los comandos de consola tienen cláusulas TO PRINTER y TO FILE que reemplazan a SET PRINTER y SET ALTERNATE. La siguiente tabla lista todas las operaciones de consola en el lenguaje Clipper.

Tabla 1.22: Operaciones de Consola.

Operación	Salida
?/??	El resultado de una o más expresiones
ACCEPT	Un prompt. Espera por la entrada de un carácter
DISPLAY/LIST	El contenido de una base de datos
INPUT	Un prompt.Espera por la entrada de cualquier tipo
LABEL FORM	El contenido de la base de datos como etiqueta
REPORT FORM	El contenido de la base de datos como reporte
TEXT..ENDTEXT	Un bloque de texto
TYPE	El contenido de un fichero
WAIT	Un prompt.Espera la entrada de un caracter
QOUT()QQOUT()	El resultado de una o más expresiones

SET CONSOLE es un comando importante cuando se usa en operaciones de consola ya que permite suprimir temporalmente la salida por pantalla sin afectar la salida a fichero e impresora. Por ejemplo, si usted quiere imprimir un reporte, éste puede desconcertarlo porque usted vé moviéndose hacia arriba por pantalla lo que se está imprimiendo. Para controlar esto, se pone SET CONSOLE OFF antes de la impresión como en el siguiente segmento de código.

```
USE Accounts NEW
SET CONSOLE OFF
REPORT FORM Accounts TO PRINTER
SET CONSOLE ON
CLOSE Accounts
```

12.2.- Operaciones de pantalla

Las operaciones de pantalla direccionan la pantalla (al mismo tiempo que la impresora) directamente, usualmente por un direccionado de una fila y columna específica. Esta operación se distingue de las operaciones de consola en que son ignoradas por el SET ALTERNATE, SET CONSOLE y SET PRINTER.

Tabla 1.23: Operaciones de pantalla.

Operación	Resultado

@...BOX	Dibuja una caja
@...CLEAR	Borra la pantalla
@...SAY	Muestra el resultado de una operación
@...TO	Dibuja una caja
CLEAR	Borra pantalla
COL()	Retorna la columna actual
ROW()	Retorna la fila actual
MAXCOL()	Retorna la cantidad máxima de columnas
MAXROW()	Retorna la cantidad máxima de filas
SAVESCREEN()	Salva una región de pantalla
RESTSCREEN()	Restaura la región de pantalla salvada

De todas las operaciones de pantalla, solo @...SAY puede redireccionar la impresora con SET DEVICE y escribir hacia un fichero con SET PRINTER. La impresión es discutida más adelante en ésta sección.

12.3.- Controlando el color de pantalla

SETCOLOR() es usada para salvar el color actual y opcionalmente poner nuevos colores para la subsiguiente impresión por pantalla. La cadena de color esta formada por diferentes colores, correspondiendo cada uno a las diferentes regiones de la pantalla.

Los colores standard son usados por todos los comandos de consola y de pantalla (con exepción de @...GET) cuando son mostrados por pantalla. Para el @...GET el lugar que ocupa el color realzado (el segundo) es usado por el área del GET y el lugar (el último) para los no seleccionados causa que todos los GETs, menos el actual, usen diferentes atributos de color.

Para más información respecto a los colores incluída la lista de colores permitidos, vea SETCOLOR() en el capítulo de funciones standard.

12.4.- Controlando el destino de la salida

En Clipper, el destino de salida por omisión es siempre la pantalla, pero también es posible direccionar la salida hacia la impresora o hacia un fichero en disco.

12.4.1- Direccionando la salida hacia la impresora

La forma de direccionar la salida hacia la impresora depende de la operación en que usted la use. Varios comandos de consola tienen la cláusula TO PRINTER que se designa para hacer la salida hacia la impresora. Por ejemplo, para imprimir el juego de etiquetas usted puede usar:

```
LABEL FORM Ship TO PRINTER
```

Para los comandos de consola que no soportan ésta cláusula, el comando SET PRINTER puede usarse. Por ejemplo:

```
SET PRINTER ON
DO WHILE .NOT. EOF()
  QOUT(Name)
  SKIP
ENDDO
SET PRINTER OFF
```

SET PRINTER puede ser usado además en lugar de la cláusula TO PRINTER si usted prefiere. Como fué declarado anteriormente usted puede desear el SET CONSOLE OFF antes de direccionar la salida de consola hacia el printer.

Para @...SAY, use el comando SET DEVICE como en el siguiente ejemplo:

```
SET DEVICE TO PRINTER
DO WHILE .NOT. EOF()
  @ 1, 1 SAY Name
  SKIP
ENDDO
SET DEVICE TO SCREEN
```

A diferencia de las operaciones de consola, @...SAY muestra hacia la pantalla o hacia la impresora pero nunca hacia ambos dispositivos simultáneamente.

Clipper permite otros controles con la impresora, incluyendo la habilitación para acceder a la actual posición del cabezal con PROW() y PCOL(). Estas funciones son usadas para un direccionamiento relativo de la impresora con @...SAY de la misma forma que ROW() y COL() son usados para el direccionamiento relativo de la pantalla.

Durante la impresión con @...SAY, se produce un cambio de página cuando la fila que usted direccionó es menor que el valor actual de PROW(). SETPRC() puede ser usada para limpiar PROW() y PCOL() evitando cambios de página en situaciones no deseadas.

SET MARGIN pone el margen izquierdo de la impresora. Con la salida de consola, el margen puesto es usado para indentar la salida donde quiera que exista una nueva línea. Con @...SAY, los márgenes puestos son adicionados al valor de columna especificado.

El comando SET PRINTER tiene una segunda forma que es usada para cambiar el dispositivo de salida, permitiendo acceder a más de una impresora y dirigir la salida hacia un fichero. SET PRINTER TO pone el dispositivo de salida para todas las salidas impresas incluyendo las operaciones de consola y de pantalla.

12.4.2.- Direccionando la salida hacia un fichero

La salida @...SAY no puede ser enviada directamente hacia un fichero. En cambio, usted puede reenrutar esta impresión hacia un fichero con SET PRINTER TO como en el siguiente ejemplo:

```
SET PRINTER TO AtOut.prn
SET DEVICE TO PRINTER
//
// < Los comandos @...SAY van aquí >
//
SET DEVICE TO SCREEN
SET PRINTER TO
```

Redireccionar la salida de la impresora hacia un fichero texto con SET PRINTER TO causa que todas las salidas vayan hacia el fichero incluyendo @...SAY. Por consiguiente, éste puede ser usado para enviar @...SAY y salidas de consola hacia el mismo fichero mientras SET DEVICE TO PRINTER y SET PRINTER ON están activos.

```
SET PRINTER TO AtOut.prn
SET DEVICE TO PRINTER
//
// < Los comandos @...SAY van aquí >
//
SET PRINTER ON
//
// < Las operaciones de consola van aquí >
```

```
//
SET PRINTER OFF
SET DEVICE TO SCREEN
SET PRINTER TO
```

Para las operaciones de consola, existen métodos para el direccionamiento de la salida hacia un fichero que no implica un reenrutamiento de las salidas a la impresora. Esta forma sencilla es usando la opción TO FILE si ésta es soportada. Si no, use el siguiente método:

```
SET ALTERNATE TO OutFile.prn
SET ALTERNATE ON
//
// < aquí van las operaciones de consola >
//
SET ALTERNATE OFF
CLOSE ALTERNATE
```

13.- Sistema de teclado

Por defecto, una aplicación desarrollada en Clipper automáticamente salva los caracteres escritos en el teclado en un buffer llamado buffer de teclado (o buffer typeahead). Los caracteres en el buffer de teclado son removidos sobre la base de que el primero que entra es el primero que sale (FIFO) y usado como entrada cuando es requerido por un estado de espera o INKEY(). Los estados de espera incluyen ACCEPT, INPUT, READ, WAIT, ACHOICE(), DBEDIT() y MEMOEDIT().

El buffer para las teclas permite al usuario continuar tecleando sin tener que esperar que la aplicación realice la captura lo cual es muy conveniente especialmente para aplicaciones que la entrada de datos es muy intensiva y el usuario es un mecanógrafo rápido. Esta característica puede no afectar nunca el modo en que usted programe su aplicación, pero puede darse el caso cuando usted desea controlar el buffer de teclado. Esta sección, describe los comandos y funciones disponibles en Clipper que afectan el buffer de teclado.

13.1.- Cambiando el tamaño del buffer del teclado.

SET TYPEAHEAD es usado para controlar el número de caracteres que puede tener el buffer del teclado. El tamaño mínimo del buffer del teclado es de cero caracteres, lo que significa que se deshabilita el buffer del teclado y permite al usuario teclear solo cuando una operación de entrada requerida se activa en ejecución. El tamaño máximo del buffer del teclado es de 32767 caracteres. Al usar el comando SET TYPEAHEAD en una aplicación, limpia el contenido actual del buffer del teclado antes de cambiar el tamaño del mismo.

13.2.- Poniendo caracteres en el buffer del teclado

KEYBOARD es usado en programación para poner caracteres dentro del buffer del teclado. Este comando limpia todas las teclas pendientes antes de situar uno o más caracteres ASCII en el buffer del teclado. La función CHR() puede ser usada con KEYBOARD para representar las teclas no imprimibles como es el caso del RETURN (CHR(13)).

Los caracteres situados de esta manera en el buffer del teclado son tratados precisamente como si ellos hubieran sido escrito desde el teclado -ellos no son ejecutados hasta que no sean extraídos por un estado de espera o por INKEY().

El uso de KEYBOARD es muy conveniente cuando desarrollamos programas de demostración auto-ejecutables en Clipper, y es comúnmente usado con la función usuario ACHOICE() para forzar la selección de un menú particular dentro de ciertas circunstancias.

Refiérase al apéndice de códigos de INKEY() al final de éste libro para una lista completa de los códigos de teclas.

13.3.- Leyendo caracteres desde el buffer del teclado

Existen tres funciones habilitadas para darle información acerca de las teclas en el buffer del teclado. Estas son INKEY(), LASTKEY(), NEXTKEY(). Cada función retorna un valor numérico entre -39 y 386, identificando el código INKEY() de las teclas incluyendo las teclas de función y las combinaciones de teclas Alt-función, Ctrl-función, Alt-letra y Ctrl-letra.

INKEY() es solo una de las funciones de teclado que actualmente extrae un caracter del buffer del teclado. En este sentido ella puede, o no, efectuar un estado de espera, para la entrada dependiendo de sus argumentos. INKEY() extrae el próximo caracter pendiente desde el buffer de teclado y retorna el valor ASCII de la tecla. Esta función es muy conveniente usarla para filtrar el teclado o provocar pausa durante la ejecución.

NEXTKEY() retorna el valor ASCII del próximo caracter sin mover éste desde el buffer de teclado.

LASTKEY() retorna el valor ASCII del último caracter extraído desde el buffer del teclado. Las entradas por teclado son extraídas desde el buffer por INKEY() o por un estado de espera. Esta función tiene diversas aplicaciones incluida la determinación de la tecla usada para terminar un READ.

13.4.- Controlando las teclas predefinidas

Existen diversos comandos y funciones que controlan la acción específica de teclas predefinidas. Esto es resumido en la próxima tabla. Además estos controles son limitados, toda predefinición de tecla puede ser completamente redefinida con SET KEY.

13.5.- Reasignando las definiciones de teclas

SET KEY permite redefinir cualquier tecla para especificar un procedimiento que va a ser ejecutado cuando la tecla es presionada durante un estado de espera. Tres parámetros, PROCNAME(), PROCLINE() y READVAR() son automáticamente pasados al procedimientos como parámetros. Como con INKEY(), las teclas estan definidas con un código ASCII.

Tabla 1.24: Comandos y funciones que controlan teclas específicas

Comando/Función	Propósito
SET ESCAPE	Conmuta la terminación del READ con ESC
SET WRAP	Conmuta la navegación circular de un menu
ALTD()	Conmuta que se invoque al DEBUG con Alt-D
READEXIT()	Conmuta terminación del READ con flecha arriba y flecha abajo.
SETCANCEL()	Conmuta la terminación de una aplicación con Alt-C.

Otro comando, SET FUNCTION, permite redefinir las teclas de funciones de la misma manera que SET KEY. Este comando, es preprocesado dentro de los comandos SET KEY y KEYBOARD causando que SET FUNCTION sustituya un SET KEY para el mismo número de tecla. Esto es compatible con versiones anteriores de Clipper lo cual mantiene separadas las definiciones del SET FUNCTION y SET KEY.

Un total de 32 teclas pueden ser activadas a la véz. Al comenzar, el sistema asume que F1 es la ayuda. Si existe un procedimiento llamado Help enlazado dentro del programa actual, será ejecutado al presionarse F1 desde un estado de espera.

Todas las definiciones de SET KEY tienen precedencia sobre las definiciones standard de teclas. Esto incluye las teclas predefinidas tales como INS y DEL que son usadas en la edición, así como las teclas ESC y ALT-C que usted puede controlar dentro de los propios comandos y funciones.

13.6.- Limpiando el buffer de teclado

Usted puede limpiar todas las entradas del buffer de teclado con CLEAR TYPEAHEAD. El uso de este comando es muy recomendable cuando usted quiere estar seguro que el usuario no deje de ver mensajes importantes. Simplemente, al mostrar un mensaje y esperar por una respuesta, éste se pasaría rápidamente, usando el próximo carácter pendiente en el buffer para la respuesta. No obstante, usando un CLEAR TYPEAHEAD antes de mostrar el mensaje asegura que no existan caracteres pendientes y fuerza al usuario a leer el mensaje antes de responder.

14.- Sistema de ficheros de bajo nivel

Diversas funciones son ofertadas para permitir la manipulación de ficheros binarios del DOS directamente desde una aplicación Clipper. Estas incluyen la capacidad de abrir ficheros binarios existentes, crear nuevos ficheros y leer y escribir hacia un fichero abierto. Esta sección describe estas funciones y sus capacidades. Note que las funciones descritas aquí permiten el acceso a ficheros del DOS a bajo nivel y los periféricos. Ellas requieren un conocimiento profundo del sistema operativo y deben ser usadas con extremo cuidado. Refiérase al capítulo de Funciones Estandards de este libro para más información de una función particular.

14.1.- Abriendo un fichero

Existen dos funciones permitidas para abrir ficheros binarios. La primera FOPEN(), abre un fichero existente mientras la segunda FCREATE(), crea un fichero nuevo, habilitándolo para su uso. Ambas funciones retornan un número que representa el handle del fichero del DOS para el fichero abierto. El handle del fichero puede ser salvado en una variable para su uso posterior por otras funciones para identificar el fichero. FCREATE() permite especificar los atributos del fichero del DOS, y FOPEN() le especifica a usted el modo de apertura del fichero.

Ya que estas funciones forman parte del tratamiento de ficheros a nivel del sistema operativo ellas no se refieren a los valores del SET DEFAULT y SET PATH. En cambio, a menos que un nombre de fichero sea identificado con el path, son asumidos la torre y el subdirectorio actuales del DOS. Además no es asumida la extensión del fichero a menos que sea explícitamente indicada.

14.2.- Leyendo desde un fichero

Una vez que un fichero binario es abierto, usted puede leer el contenido del mismo con una de las dos funciones, FREAD() o FREADSTR(). Ambas funciones requieren que usted identifique el fichero usando el número del handle del mismo. Es necesario especificar el fichero en particular ya que más de un fichero binario puede estar abierto al mismo tiempo.

FREAD() y FREADSTR() son muy similares funcionalmente. La principal diferencia es que FREAD() requiere que se especifique donde van a ser salvados los caracteres leídos desde el fichero, y FREADSTR() retorna los caracteres leídos como una cadena de caracteres.

En ambos casos, no obstante, los datos leídos en el fichero binario son en forma binaria. Por consiguiente varias funciones son ofertadas para convertir un dato binario a numérico, para que usted pueda manipular su información en su aplicación Clipper.

14.3.- Escribiendo hacia un fichero

Una vez que el fichero binario es abierto, usted puede escribir en él con FWRITE(). Esta función requiere que usted identifique el fichero usando el número del handle del fichero. Especificar un fichero particular es necesario ya que pueden estar abiertos más de un fichero al mismo tiempo. Usted no podrá usar FWRITE() en un fichero que fué abierto en modo solo-lectura (read-only). Con FWRITE() usted especifica una cadena de caracteres que será escrita hacia el fichero. Varias funciones son ofertadas en Clipper para convertir datos numéricos a forma binaria.

14.4.- Manipulando el puntero del fichero

Cuando un fichero binario es abierto para su uso, el puntero del fichero es posicionado al inicio del mismo. Las funciones de lectura y escritura mueven el puntero del fichero, según se requiera, al efectuar estas operaciones.

Otra función, FSEEK(), permite el movimiento del puntero directamente a una posición específica en un fichero. Esta función es particularmente usada si usted necesita la lectura o escritura de caracteres en algún lugar de la mediación del fichero pero no le interesa la información que precede a estos. En vez de efectuar una operación de lectura sobre datos indeseados, usted puede utilizar un FSEEK() lo cual es mucho más eficiente para mover el puntero hacia la posición correcta dentro del fichero. Entonces, usted podrá leer o escribir los caracteres necesarios.

FSEEK() está habilitada de manera muy flexible para posicionar el puntero del fichero. Usted le especifica a la función un argumento que dice dónde comienza, desde el inicio, desde la posición actual, o desde el final del fichero, y otro argumento que indica el número de bytes a mover.

14.5.- Cerrando ficheros

FCLOSE() cierra un fichero binario y escribe todo lo asociado a éste en el buffer del DOS en el disco. Esta función requiere que usted identifique el fichero a cerrar mediante el número del handle del fichero. Es necesario especificar un fichero en particular ya que muchos ficheros binarios pueden estar abiertos al mismo tiempo.

14.6.- Detección de errores

Cuando usamos funciones de ficheros a bajo nivel, usted puede chequear errores a través del valor individual de retorno de cada función o con la función FERROR(). El método a usar depende de que nivel de detalle usted desea relativo al error. La siguiente tabla resume el valor de retorno de las funciones de fichero de bajo nivel que indican una condición de error.

Tabla 1-25: Condiciones de error de las funciones de bajo nivel

Función	Condición de error
FCLOSE()	Retorna falso (.F.)
FCREATE()	Retorna -1
FOPEN()	Retorna -1
FREAD()	Retorna 0 o un valor menor que el número de bytes a leer
FREADSTR()	Retorna cadena nula
FSEEK()	No brinda condición de error
FWRITE()	Retorna 0 o un valor menor que el número de bytes a escribir

FERROR retorna el número de error del DOS para la última función de fichero de bajo nivel ejecutada. Si no existe error, FERROR() retorna 0. Después de determinar que una función a fallado, mediante el chequeo de su valor de retorno, usted puede usar FERROR() para delimitar la causa del error.

15.- El sistema Browse

Un nuevo sistema Browse ha sido implementado en Clipper 5.0 el cual usa el tipo de dato objeto. El sistema Browse fue echo con dos tipos de clases: TBrowse y TBColumn las cuales son discutidas en esta sección. Para una lista completa de las variables de instancia exportadas y métodos, refiérase al capítulo de Clases Estandard de este libro.

15.1.- La clase TBrowse

15.1.1.- ¿Qué es un objeto TBrowse?

Un objeto TBrowse es un mecanismo para "browsing" de propósito general para una tabla de datos orientada. El objeto TBrowse provee una sofisticada arquitectura para la adquisición, formateo y muestreo de datos. La recuperación de datos y el posicionamiento en el fichero son efectuados a través de bloques de código suministrados por el usuario. El formateo por pantalla es completamente controlable mediante el uso de formateos de cadenas especificados por el usuario.

Un objeto TBrowse evalúa un bloque de código suministrado para la recuperación de datos. Los datos son organizados dentro de filas y columnas y mostrados dentro de los confines de un área rectangular especificada en la pantalla. Esta conducta de programación permite el recorrido por los datos. Nuevos datos son automáticamente recuperados a través de éste recorrido. Si el cursor es movido después del rectángulo visible, cualquier fila o columna que exista después es mostrada automáticamente.

15.1.2.- Creando un objeto TBrowse

La sintáxis para crear un objeto TBrowse es la siguiente:

browse := TBrowseNew(<nTop>, <nLeft>, <nBottom>, <nRight>)

Esta línea de código crea un nuevo objeto TBrowse y lo salva dentro de la variable browse. Las coordenadas definen el área rectángular para el objeto TBrowse.

Una función adicional de la clase TBrowse es TBrowseDB(). Esta función tiene una sintáxis similar a TBrowseNew pero es designada para su uso cuando "browsing" un fichero base de datos. TBrowseDB crea un objeto TBrowse con un bloque de códigos por defecto para manipular una base de datos. El bloque de códigos por defecto ejecuta las operaciones GO TOP, GO BOTTOM y SKIP.

15.1.3.- Definiendo columnas

Las columnas de un objeto TBrowse consisten en objetos TBColumn. Cada objeto TBColumn contiene la información necesaria para definir una sola columna de un browse (Ver la sección Las Clases TBColumn abajo).

En el siguiente ejemplo, un nuevo objeto TBColumn es creado y adicionado a un objeto browse ya existente.

```
column := TBColumnNew( "Name", { || name } )
browse:addColumn( column )
```

La función TBColumnNew() crea un nuevo objeto de columna. El mensaje TBrowse:addColumn() es entonces quien envia al objeto TBrowse a adicionar una nueva columna dentro del browse. Las columnas pueden ser adicionadas, modificadas, o reorganizadas en cualquier momento.

15.1.4.- Definiendo colores

Un objeto TBrowse usa una tabla de color para controlar los atributos de pantalla de los datos mostrados. Una tabla de color es una lista ordenada de especificaciones de colores Clipper. Por omisión un objeto TBrowse usará la tabla de colores como los cinco colores definidos por la función standard de colores SETCOLOR(). Si es deseado, una tabla compleja de colores puede ser especificada asignándosele como valor a TBrowse:colorSpec.

Existen tres modos de cambiar el color al objeto TBrowse:

* Suministrando una nueva tabla de color usando TBrowse:colorSpec. Esto afecta el color de todas las celdas.

* Asignando TBColumn:defColor a uno o más objetos TBColumn. Esto afecta el color de una columna entera, incluyendo encabezamiento y pie (ver la sección
Clase TBColumn).

* Asignando un bloque de códigos de control de color a través de TBColumn:colorBlock para uno o más objetos TBColumn. Esto permite el color de valores de datos individuales para ser controlados (ver la sección Clase TBColumn).

Todos los colores son especificados usando el índice de la tabla de color en uso por el objeto TBrowse. Si no se define la tabla de color usando TBrowse: colorSpec, el color se buscará limitado de 1 a 5, correspondiendo los cinco colores a los de la función standard SETCOLOR().

15.1.5.- Estabilización

Las operaciones necesarias para adecuar la visualización de los datos del TBrowse incluyen el posicionamiento del dato fuente, actualización del dato (vía bloque de códigos suministrado en el objeto TBColumn), formateo del dato y actualización física del display. Estas operaciones son referidas colectivamente como estabilización del objeto TBrowse. Cuando un TBrowse es estable, ello significa que todas las filas y columnas estan correctamente mostradas, que la base de datos u otra fuente de datos, esta posicionada en la fila actual y que la celda actual esta resaltada. Para evitar las demoras de respuestas debido a las operaciones potencialmente largas de recuperación de datos, la estabilización ocurre en pequeños incrementos. El método de estabilización ejecuta una pequeña parte del proceso de estabilización cada vez que éste es invocado. Este retorna verdadero (.T.) si la estabilización es completada; de lo contrario retorna falso (.F.). Esto permite a los procesos de estabilización ser interrumpidos en respuesta a una tecla presionada. El mensaje TBrowse:stabilize() es usualmente enviado en un ciclo semejante a éste:

```
DO WHILE ( .NOT. browse:stabilize() ) ; ENDDO
```

El ejemplo anterior fuerza la completa estabilización del objeto TBrowse. Como se mencionó anteriormente, de todos modos, es regularmente deseable permitir al usuario interrumpir la estabilización. Esto es hecho para chequear simplemente la entrada por teclado durante la estabilización del ciclo:

```
DO WHILE ( .NOT. browse:stabilize() )
  IF (NEXTKEY() != 0 )
    EXIT
  ENDIF
ENDDO
```

Esta técnica permite al usuario moverse a la próxima pantalla de datos sin esperar porque la actual pantalla sea llenada.

15.2.- La clase TBColumn

15.2.1.- Qué es un objeto TBColumn?

Un objeto TBColumn es un simple objeto que contiene información necesaria que define completamente una columna en el objeto TBrowse (ver la sección sobre La Clase TBrowse). Los objetos TBColumn no tienen métodos, solo variables de instancia exportadas.

15.2.2.- Función Clase

La función asociada con la clase TBColumn es como sigue:

TBColumnNew(<cHeading>, <bBlock>)

TBColumnNew() retorna un nuevo objeto TBColumn con los valores especificados por TBColumn:heading y TBColumn:block. Otros elementos del objeto TBColumn pueden ser asignados directamente usando la sintáxis para la asignación de variables instancias exportadas.

16.- El Sistema Get

Un nuevo sistema Get ha sido implementado en Clipper 5.0 el cual usa el tipo de dato objeto. El sistema Get es solo una clase ficticia llamada Get la cual es discutida en esta sección. Para una lista completa de las variables instancias exportadas y los métodos, refiérase al capítulo de Clases Standard en este libro.

16.1.- Implementación de @...GET y READ

16.1.1.- @...GET

El pre-procesador traslada cada comando @...GET dentro del código Clipper hacia un nuevo objeto GET conteniendo las características especificadas en @...GET y adiciona el objeto al arreglo llamado GetList.

Una sutileza importante de éste proceso es que además del predefinido arreglo público GetList, pueden ser además creados y usados arreglos privados nombrados GetList . A partir de lo anterior, se permite el anidado y uso local del subsistema @...GET/READ. Igualmente importante es el uso de los objetos Gets en distintas formas a partir de los comandos standards @...GET y READ.

16.1.2.- READ

Como resultado de uno o más comandos @...GET el arreglo GetList contendrá uno o más objetos Gets en el momento que un comando READ es usado. El procesador traslada el comando READ dentro de una llamada hacia la función READMODAL(), con la variable GetList especificada como argumento.

Dentro de READMODAL(), el proceso READ es manipulado de la forma que usted podia esperar semejante a la forma que son recuperadas las teclas con INKEY(),y cada tecla presionada provoca una acción particular dentro del sistema standard GET. Cuando un proceso READ es finalizado READMODAL() termina retornando NIL.

16.1.3.- Getsys.prg

La función READMODAL() es definida en el módulo del sistema Getsys.prg. Este módulo define las otras funciones y servicios que son necesarias para implementar el subsistema @...GET/READ así como el funcionamiento standard de los objetos Gets.

Por ejemplo, las funciones de validación las cuales muestran mensajes sobre el sistema SCOREBOARD, la asignación que determina que Esc pueda salir de un READ, y el servicio que conmuta el estado de la tecla Ins estan definidos en Getsys.prg.

16.2.- La variable GET

Cada objeto Get tiene una variable GET asociada con éste. En el siguiente ejemplo var es la variable GET cuyo valor es actualizado por la edición interactiva que tiene lugar durante un READ.

@ nRow, nCol GET var

Considere esta descripción simplificada de que le pasa a un objeto GET durante
una secuencia @...GET/READ:

1.- Mostrar -el objeto Get recupera el valor de la variable Get y copia ésta dentro de un buffer que puede ser editado. Este buffer es mostrado en la pantalla.

2.- Edición -cuando el GET es seleccionado el objeto Get es enviado y un mensaje es dado a su foco de entrada. Un cursor es entonces situado dentro del buffer de edición. Las teclas del usuario son interpretadas como comandos de edición, y el contenido del buffer de edición es manipulado en correspondencia (por ejemplo, el borrar caracter e insertar un caracter).

3.- Asignación -cuando se indica por algún evento de edición (por ejemplo, salvar y salir, implicado por Ctrl-W), el objeto Get traslada el contenido del buffer de edición dentro de un valor del mismo tipo que el de una variable GET y fija la variable con este valor.

Note que, en estos pasos el objeto Get ejecuta todas las acciones. La transformación del valor de una variable GET dentro del buffer de edición, la manipulación del contenido del buffer de edición, y el paso desde el buffer hacia la variable GET son todas acciones hechas por el objeto Get basadas en mensajes recibidos por éste.

Por tanto un objeto Get es una conexión explícita con una variable GET, cuyo valor es buscado antes de editar y fijado después de editar. Es importante examinar cómo esta conexión es implementada.

En términos concretos, que es necesario a la hora de actualizar y asignar a una variable dada. En el lenguaje Clipper, esto se hace mejor definiendo un bloque de códigos que, cuando es ejecutado usando la función EVAL(), actualiza y fija el valor de una variable GET. Los bloques son definidos como sigue:

```
{ || getVar }                  // recuperar
{ |newVal| getVar := newVal }  // fijar
```

donde "getVar" es la variable GET y "newVal" es el contenido del buffer de edición del objeto Get después de la edición. Estos bloques son condensados para más eficiencia dentro de un solo Get y el bloque queda:

```
{ |newVal| IF ( newVal == NIL, getVar, getVar := newVal ) }
```

Cuando es evaluado con EVAL(), éste bloque asigna a "getVar" "newVal" si un valor no-NIL en "newVal" es suministrado como argumento; de lo contrario, éste simplemente retorna el valor actual de "getVar". Este bloque de códigos es el único que conoce el objeto Get de su variable GET asociada. En efecto, el término variable GET puede ser expresado más completamente como la variable referenciada dentro del bloque GET.

16.3.- Focos de entrada

Un objeto Get existe en uno de dos estados: Con foco de entrada o no. Cuando un objeto Get tiene foco de entrada, el foco de entrada es el Get seleccionado en el actual READ.

Los focos de entrada son dados y tomados fuera desde un objeto Get por la función READMODAL(), la cual ejecuta el comando READ. Inicialmente READMODAL() da focos de entrada al primer elemento en el arreglo GetList que ésta recibe. Entonces, por ejemplo, si la tecla de cursor abajo es oprimida, READMODAL() toma focos de entrada desde el Get y envia éste hacia el próximo GET en el arreglo.

El objeto Get puede ser editado solo mientras tiene focos de entrada. De lo contrario el GET es para mostrar solamente. Esto tiene diversas aplicaciones.
* Algunas de las variables instancias pertenecientes a un objeto Get tienen solo significado mientras el objeto Get tenga foco de entrada. Estas son típicamente variables instancias que tienen que ver solo con la edición. Ellas son:

. Get:buffer edición del buffer
. Get:decPos la posición del punto decimal dentro del buffer de edición
. Get:original el valor de la variable GET en el momento que el objeto Get estuvo dando foco de entrada.
. Get:pos posición del cursor

Si estas variables instancias son referidas mientras el objeto Get no tiene foco de entrada, el valor retornado es NIL. Cualquier intento hecho para asignar (ejemplo Get:pos := 5) mientras el Get no tiene foco de entrada es ignorado.

* Casi todos los métodos de un objeto Get estan solo presentes mientras el objeto Get tiene foco de entrada. Esto ocurre porque casi todos los métodos del GET estan implementados con algún aspecto de los procesos de edición (ejemplo, manipulación del buffer de edición o validación). Debido a esto, los métodos que están presentes cuando el GET no tiene foco de entrada son los siguientes:

. Get:display() muestra el objeto Get
. Get:setFocus() da foco de entrada al objeto Get

Si cualquier otro método es invocado mientras el GET no tiene entrada de foco, el método no tiene efecto. En cualquier caso un error, nunca es generado.

Nota: En sistemas más orientados a objeto el concepto de foco de entrada es mucho más general que éste en Clipper 5.0.

16.4.- La edición del buffer

El propósito primario del objeto Get es la edición del valor de la variable GET asociada. Durante la edición, ésta aparece como si su valor estuviera siendo contínuamente alterado, lo cual no ocurre así. En cambio, el valor inicial de la variable GET es mapeado dentro de un buffer de edición y durante el curso de ésta los cambios son hechos dentro del buffer. Solo cuando los eventos destinados a la salva del valor editado se producen, es cambiada la variable GET el contenido del buffer de edición es trasladado con el tipo de valor apropiado hacia la variable GET.

Existen diversas razones importantes para usar este esquema:

* El sistema GET/READ requiere que los cambios de edición sean descartados bajo ciertas circunstancias (ejemplo, cuando el mensaje Get:undo() es enviado).

* Valores de un tipo diferente que caracteres no son editables en su forma natural asi que todos los valores deben ser transformados a tipo de caracteres para la edición.

* Similarmente, las intrucciones de formateo (como las que son encontradas en la claúsula PICTURE) no tienen objetivo fuera del contexto de un buffer de caracteres. En efecto una claúsula PICTURE puede aparecer a pesar de que se especifique cómo trasladar el valor de la variable GET hacia un buffer de edición.

El hecho que durante la edición, un buffer (que puede ser la misma variable GET) sea editado, trae consigo otro punto: mientras el objeto Get tenga foco de entrada, es el buffer de edición no el valor actual de la variable GET el que se muestra en pantalla.

Esta es una distinción importante. Si el mensaje Get:display() es enviado hacia un objeto Get cuando el GET no tiene foco de entrada, el valor de la variable GET es mostrado. Si Get:display() es enviado mientras el GET tiene foco de entrada, el buffer de edición es mostrado. Esto es así por razones obvias que, mientras el GET tenga foco de entrada, los cambios hechos dentro del buffer de edición pueden ser visibles.

Como se discutió previamente, el valor de la variable GET es trasladado antes de la edición, hacia el buffer de edición de caracteres. Entonces, después de la edición, el buffer de edición es trasladado como un valor que es fijado en la variable GET. Estas son las dos interacciones que tienen lugar entre el buffer de edición y la variable GET. A continuación viene una lista de métodos en los que estas interacciones tienen lugar.

* Get:setFocus()

Entre otras cosas que suceden cuando el objeto GET recibe foco de entrada, el buffer de edición es creado y el valor de la variable GET es mapeado dentro de él.

* Get:assing()

Este mensaje, que típicamente se envia después de la edición, traslada el contenido del buffer de edición en un valor del tipo apropiado el cual es asignado a la variable GET.

* Get:updateBuffer()

Este método reinicializa el buffer de edición desde la variable GET. Esto es, el valor de la variable GET es nuevamente trasladado y mapeado dentro del buffer de edición. El mensaje Get:updateBuffer() es típicamente enviado en cualquier situación donde el valor de la variable GET pueda ser cambiado fuera del sistema GET por ejemplo cuando un proceso SET KEY es invocado dentro de un READ.

* Get:reset()

En términos de interacción entre la variable GET y el buffer de edición, éste método es la misma cosa que Get:updateBuffer(). Sin embargo su propósito principal es reinicializar el mecanismo de edición del GET. Get:reset() remapea el valor de la variable GET dentro del buffer de edición y restaura el cursor de edición hacia su posición inicial.

La cuestión importante a recordar alrededor del esquema de edición de los objetos GET es que desde que un buffer reservado es usado para la edición actual, la actualización de la variable GET debe estar explícitamente contenida en las diversas situaciones descritas arriba. Para ejemplos de programación de esta situación vea la definición de la función READMODAL() en GetSys.prg.

17.- Comandos de edición

La edición actual de un GET es hecha por el envio de varios mensajes comandos de edición hacia el objeto GET.

Es extremadamente importante distinguir entre los propios comandos de edición y las teclas que provocan acciones dentro del sistema GET estandar. Hablando claramente, el único conocimiento que tiene el sistema de la conexión entre una tecla particular y un comando de edición particular es en la función READMODAL().

Como un ejemplo, considere la asociación estandar entre la tecla backspace y el método Get:backspace(). Si, durante la edición, la tecla backspace es leída, READMODAL() envia el mensaje Get:backspace() al objeto GET durante la edición actual. El método Get:backspace() por sí mismo no tiene conocimiento de la tecla que lo provocó; éste simplemente ejecuta el borrado y el movimiento del cursor.

Esto significa que la asociación estandar entre las teclas particulares y los comandos de edición particulares pueda ser completamente remapeada simplemente cambiando el código en la función READMODAL(). En efecto, no es necesario que los mensajes de edición sean específicamente provocados por todas las teclas.

18.- Validación

La validación proporcionada en el sistema GET y soportada por el objeto GET es dividida en tres categorias: validación pre-edit, post-edit e in-edit.

18.1.- Validación pre-edit

La validación pre-edit determina si el objeto GET va a tener foco de entrada. En código Clipper, la validación pre-edit es especificada por la claúsula WHEN en algún comando @...GET. Si la claúsula WHEN esta presente en un @...GET en particular, la expresión dada es almacenada como un bloque de código en Get:preBlock.

Durante un READ (esto es, llamada a un READMODAL()), previo a que a el GET le sea dado foco de entrada, el bloque almacenado en Get:preblock es ejecutado usando EVAL(). Si el resultado es falso (.F.), el GET no será editado y es soslayado por READMODAL(). Si el resultado es verdadero (.T.), el GET será editado y la operación procede normalmente.

18.2.- Validación Post-edit

La validación Post-edit determina si, después de haber sido editado, el valor de la variable GET es legal. Si es así, el foco de entrada puede ser tomado desde el GET, y el READ procede apropiadamente. Si no, el objeto GET será reeditado hasta que el valor de la variable GET sea legal.

Tres modos de la validación Post-edit pueden ocurrir en un sistema GET.

* Una condición especificada por la claúsula VALID en un comando @...GET. Si una claúsula VALID existe, la expresión dada es almacenada como un bloque de código en Get:postBlock.

Durante un READ, cuando un fin de edición es recibido (esto es, cuando READMODAL() se mueve al próximo GET o sale del READ), Get:postBlock es ejecutado usando EVAL(). Si el resultado es falso (.F.), la edición debe continuar. Si el resultado es verdadero (.T.), el proceso READ procede.

Note que, ya que la expresión de validación esta referida en la propia variable GET, el contenido del buffer de edición debe ser trasladado en un valor y asignado a la variable GET antes que Get:postBlock sea actualmente evaluado.

* Un rango numérico especificado por la claúsula RANGE en un comando @...GET. Una claúsula RANGE es actualmente trasladada dentro de una claúsula VALID de la forma:

```
... VALID RangeCheck( lower, upper )
```

donde RangeCheck() es una función definida en GetSys.prg, la cual implementa la conducta definida en la claúsula RANGE (por ejemplo, el mensaje SCOREBOARD).

* La validación de fecha es automáticamente proporcionada para cualquier objeto GET cuya variable GET es de tipo fecha, y el GET es prevenido con la pérdida del foco de entrada si la fecha es inválida. Es importante notar que la porción genérica de esta tarea (esto es, determinar si el valor de fecha es válido o no) es implementado por Get:badDate, mientras la porción específica del sistema GET es implementada por la función DateMsg() definida en GetSys.prg.

18.3.- Validación in-edit

La validación in-edit consiste en que dato puede entrar en el buffer de edición durante el proceso de edición. Típicamente un intento de entrar un dato inválido es simplemente ignorado. Existen varios tipos de validación in-edit:

* Restricción basada en el tipo. son restricciones automáticas proporcionadas por el sistema GET que son basadas en el tipo de la variable GET. Por ejemplo, los caracteres alfabéticos no pueden ser insertados dentro de un buffer de edición de un GET numérico.

* Restricción picture. restricción definida por la claúsula picture del objeto GET.