39
2ej

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES

"A R A G O N"

ELEMENTOS PRACTICOS Y TECNICAS DE PROGRAMACION AVANZADA EN CLIPPER 5.01

T E S I S

Que para obtener el Título de:

INGENIERO EN COMPUTACION

P r e s e n t a n

FERNANDO TECUAPACHO TEXIS

JORGE ALFREDO CASTAÑO SALAZAR

Asesor: Ing. Luis L. Jiménez García

San Juan de Aragón, Edo. de Méx. 1994

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

Dedicatoria

A Ti señor, por darme todas aquellas cosas que han hecho de este trabajo la culminación de una etapa más de mi vida y el comienzo de una nueva meta.

A mis padres, con todo cariño. Quienes toda su vida se han preocupado por mi; a quienes les debo todo lo que soy.

Los quiero mucho.

Fernando Tecuapacho Texis

A mis padres, porque gracias a su amor y orientación he logrado una meta más en la vida.

A mis hermanos, en especial a mi hermana Angélica; por que han sido mis grandes compañeros.

A mis amigos, en especial a Rocío, Esthela, Jorge y Enrique: por toda su ayuda y por la confianza depositada en mi.

Jorge Alfredo Castaño Salazar

Agradecimientos

A La Escuela Nacional de Estudios Profesionales "Aragón" de la Universidad Nacional Autónoma de México por haberme abierto las puertas hacia el conocimiento y a la formación profesional.

Al Ing. Luis L. Jimenez García, por sus valiosas observaciones y consejos, que me permitieron presentar este trabajo.

Al Ing. Horacio Loyo Miranda. por haberme motivado y brindado su apoyo, haciendo posible, de esta manera, la terminación del presente trabajo.

A todos mis profesores y a mi patria gracias.

Fernando Tecuapacho Texis

A mis hermanos, por su apoyo y por ser mis mejores amigos.

A mis profesores de la U.N.A.M., tanto de la Escuela Nacional Preparatoria num. 3, cómo de la Escuela Nacional de Estudios Profesionales Aragón

Al Ing. Luis Lorenzo García Jiménez, por su confianza depositada en nuestro trabajo.

A mi amigo y compañero de tesis, Fernando.

Jorge Alfredo Salazar

CONTENIDO

INTRODUCCION 1

1. EL COMPILADOR Y SU AMBIENTE DE TRABAJO

1.1 Clipper como lenguaje de desarrollo 5

1.2 Requerimientos de hardware, software y especificaciones técnicas 8
1.2.1 Requerimientos de hardware y software 8
1.2.2 Especificaciones técnicas 9
1.2.3 Características generales 10

1.3 El compilador 10
1.3.1 Traductores 10
1.3.2 Opciones del compilador 12

1.4 Variables de ambiente 16

2. EL ENCADENADOR RTLINK, LIBRERIAS Y HERRAMIENTAS MAKE

2.1 El encadenador RTLINK 23
2.1.1 Que es el encadenador RTLINK 23
2.1.2 Sintaxis y Llaves RTLINK 24
2.1.3 Como se invoca a RTLINK 27
2.1.4 Como se configura a RTLINK 29
2.1.5 RTLINK y el DOS ERRORLEVEL 30
2.1.6 Overlays de Clipper 5.01 30

2.2 Librerías de Clipper 5.01 41

2.3 Librerías preencadenadas 42
2.3.1 Como crear librerías preencadenadas 42
2.3.2 Como preencadenar sus propias librerías 44

2.4 Herramientas make 45
2.4.1 La utilería RMAKE 45
2.4.2 La opción MAKEPATH 46
2.4.3 El proceso de comparación de RMAKE 46

3. DECLARACION DE VARIABLES Y TIPOS DE DATOS

3.1 Atributos de las variables 53
3.1.1 Tipos de datos 54
3.1.2 Visibilidad de variables 56
3.1.3 Vida de variables 59

3.2 Inicialización de las variables 60

3.3 Valores Nil 61

3.4 Operadores y expresiones 62

4. EL PREPROCESADOR DE CLIPPER

4.1 Que es el preprocesador de Clipper 5.01 67

4.2 Directivas del preprocesador 68
4.2.1 Directivas #command y #translate 68
4.2.2 Directiva #define 71
4.2.3 Directiva #ifdef 73
4.2.4 Directiva #ifndef 74
4.2.5 Directiva #include 74
4.2.6 Directiva #undef 76
4.2.7 Directiva #error 76

4.3 Técnicas de manejo 77
4.3.1 Constantes simbólicas 77
4.3.2 Inclusión de archivos 78
4.3.3 Macros del compilador 78
4.3.4 Compilación condicional 79
4.3.5 Comandos y funciones definidos por el usuario 79

5. PROCEDIMIENTOS Y FUNCIONES DEFINIDOS POR EL USUARIO

5.1 Ventajas y aplicaciones de funciones de usuario 81

5.2 Funciones y procedimientos 82

5.3 Parámetros 85

5.4 Reglas para el desarrollo de funciones 91

5.5 Biblioteca de funciones 91

5.6 El manejador de librerías LIB 93

6. ARREGLOS

6.1 Definición de arreglos y métodos de inicialización 98
6.1.1 Inicialización de arreglos 101

6.2 Funciones de Clipper relacionadas con arreglos 103

6.3 Técnicas de manejo de arreglos 113

7. CODE BLOCKS Y MACROS

7.1 Las macros y las expresiones extendidas 116
7.1.1 Que son las macros 116
7.1.2 Que son las expresiones extendidas 117

7.2 Que son los Code Blocks 118
7.2.1 Función EVAL() 119
7.2.2 Función AEVAL() 119
7.2.3 Función DBEVAL() 121
7.2.4 Otras funciones para trabajar con Code Blocks 122

7.3 Ventajas de los Code Blocks sobre las macros ... 123

7.4 Funciones de Code Blocks 126

8. LA CLASE TBROWSE Y TBCOLUMN

8.1 Conceptos 129
8.1.1 Sintaxis para creación y manipulación de objetos 130

8.2 La clase TBCOLUMN 131

8.3 La clase TBROWSE 133
8.3.1 Funciones de creación 133
8.3.2 Métodos del movimiento del cursor 136
8.3.3 Métodos diversos 138
8.3.4 Creación y ejecución de un browse 140

8.4 Técnicas de manejo para la clase TBROWSE 143
8.4.1 Creación del objeto TBROWSE sin importar la estructura del archivo .DBF 143
8.4.2 Desplazamiento de columnas 145
8.4.3 Filtrado de registros 147

9. PROGRAMACION EN AMBIENTE DE RED

9.1 Manejo de archivos en red de área local 152

9.2 Técnica y filosofía 154

9.3 Acceso compartido y exclusivo 156

9.4 Comandos y funciones para ambiente multiusuario 156

10. INTERFASE CON LENGUAJE C Y ENSAMBLADOR

10.1 El sistema extendido de Clipper 175
- 10.1.1 Las funciones y macros del sistema extendido 176
- 10.1.2 Obtención y retorno de parámetros 178
- 10.1.3 Compilación y encadenado con lenguaje C y ensamblador 180
  - 10.1.3.1 Compilación y encadenado con lenguaje C 180
  - 10.1.3.2 Construcción de librerías con lenguaje C 181
  - 10.1.3.3 Compilación y encadenado con lenguaje ensamblador 182

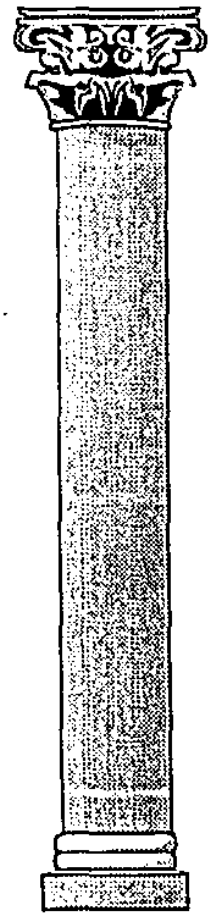
10.2 Creación de funciones en lenguaje C 183
- 10.2.1 Función para crear nombres de archivo únicos 183
- 10.2.2 Función para cambio de colores 184
- 10.2.3 Programa para ejemplificar el uso de las funciones hechas en lenguaje C 186

10.3 Creación de funciones en ensamblador 188
- 10.3.1 Función para obtener el nombre de la unidad actual 188
- 10.3.2 Función para convertir de minúsculas a mayúsculas 190
- 10.3.3 Programa para ejemplificar el uso de las funciones hechas en ensamblador 192

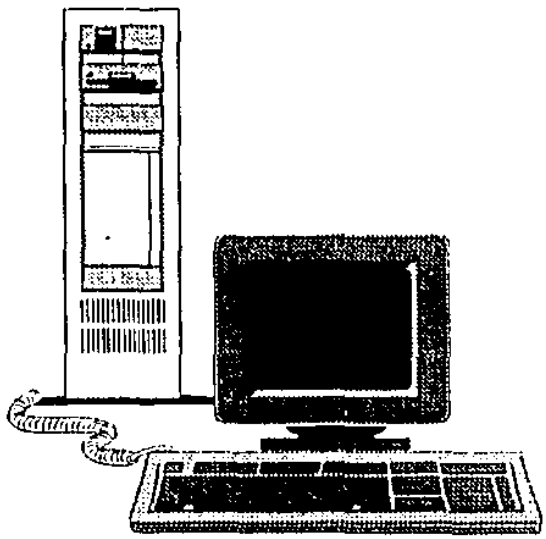
CONCLUSIONES 194

BIBLIOGRAFIA 196

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO



▪ INTRODUCCION



ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES
A R A G O N

Podemos definir a los lenguajes de programación como aquellos códigos de comunicación que sirven al hombre para decirle a la máquina lo que está debe de hacer y cómo debe de hacerlo. Constan de un conjunto de expresiones y de una sintaxis que permite agruparlas en sentencias lógicas.

El lenguaje más básico y el único que directamente entiende la computadora es el que denominamos lenguaje máquina. Dicho lenguaje está formado sólo por dos símbolos básicos: unos y ceros, por lo cual resulta muy complejo para el programador o para que personas con poca experiencia puedan comprenderlo. En lenguaje máquina se indica a la computadora exactamente todas las cosas que debe hacer y como tiene que hacerlas.

El problema se complica aún más debido a que cada procesador tiene una estructura física distinta y, por tanto, las instrucciones en lenguaje máquina adecuadas para una computadora, no son adecuadas para otra.

Con el fin de resolver algunos de estos problemas aparecieron los lenguajes ensambladores donde ya existe un cierto nivel de simbolismo, es decir, donde una instrucción simboliza o abrevia una serie de operaciones básicas. De todas formas, los lenguajes ensambladores casi lo único que vienen a despejar es la confusión de unos y ceros, ya que por lo demás siguen estando atados a cada procesador y su estructura sigue centrándose en indicar a la máquina no sólo lo que debe hacer sino también como debe hacerlo.

La resolución a estas dificultades la proporcionan los lenguajes simbólicos de alto nivel. Estos lenguajes son casi conversacionales donde ya el "como" ha de hacerse una operación casi está eludido, de modo que lo único que interesa es decir a la máquina lo que debe hacer.

El paso de lo que le decimos a la máquina que haga al como debe de hacerlo lo realizan los interpretes o los compiladores. Estas son las dos formas distintas que puede tener un lenguaje simbólico.

Un interprete es un programa traductor que examina cada una de las instrucciones y las traduce a lenguaje máquina y este proceso se ejecuta

de instrucción en instrucción y se repite cada vez que ejecutamos el programa.

En los compiladores partimos de una serie de analizadores semánticos y sintácticos que toman en bloque el programa y lo traducen a lenguaje máquina, dándonos como resultado un módulo distinto al escrito por nosotros y es el que podemos ejecutar.

Clipper en un principio fue un compilador enfocado a hacer ejecutable código escrito en dBASE III +, un manejador de bases de datos muy popular, ya que tenía una excelente capacidad de estructuración, un desarrollo procedural y modularidad. Y además presentaba como una fuerte ventaja frente a los lenguajes de alto nivel tradicionales (BASIC, FORTRAN, COBOL, PASCAL, etc.) el manejo de archivos de bases de datos, que era exclusivo del mundo de los grandes sistemas.

A pesar del manejo de bases de datos el lenguaje de dBASE posee notables deficiencias frente a los lenguajes simbólicos de alto nivel:

- Es un lenguaje interpretado, por lo cual su ejecución es lenta.
- Carece del manejo de ventanas
- No permite gestionar el modo gráfico de la pantalla.
- Solo maneja archivos de variables de memoria y de datos, pero no posee archivos de texto, ni arreglos, etc.

Clipper en su inicio dio una solución a la mayor parte de estos problemas pero en la actualidad su última versión va más allá de todo esto, por lo cual ya no debe ser considerado simplemente como "compilador de dBASE III +" sino como un lenguaje de alto nivel orientado al manejo de archivos

Desaprovechar Clipper usándolo cómo un simple compilador para dBASE III +, es desperdiciar el enorme potencial que proporciona en su versión 5.01, ya que tiene una nueva filosofía, además incluye nuevas y poderosas herramientas.

Nuestra propuesta es que usando Clipper 5.01 y versiones posteriores se puede realizar una programación seria y profesional, mediante la utilización de elementos prácticos y técnicas avanzadas, algunas de las cuales se explican a lo largo de la tesis.

En el primer capítulo se analizan las características del compilador, tales como son los requerimientos de hardware y software y especificaciones técnicas. También se ven las opciones para compilar un simple programa ó todo un sistema de acuerdo las necesidades especificas y se dan algunos ejemplos de las diferentes formas de compilación. Y en la ultima parte de capitulo se analizan brevemente las variables de ambiente que sirven para controlar el ambiente del DOS.

En el capítulo 2 "El encadenador Rtlink, Librerías y Herramientas Make", se explican las principales características y bondades del nuevo encadenador que proporciona Clipper, como son las librerías preencadenadas y el manejo de overlays, así como algunas técnicas para obtener de él un mayor provecho.

En el capítulo 3 se hace hincapié de lo importante que son las variables de memoria y los atributos que tienen, ya que podremos desarrollar aplicaciones más eficientes si tenemos un manejo adecuado de las mismas. En este mismo capítulo también se menciona la inicialización de las variables y de los nuevos operadores que son heredados del lenguaje C y Pascal.

En el capítulo 4 "El Preprocesador de Clipper" analiza brevemente el preprocesador de Clipper y se explican algunas de las técnicas de manejo, como por ejemplo el uso de Constantes Manifiestas y Comandos Definidos por el Usuario.

En el capítulo 5 se analizan las ventajas y beneficios de usar Funciones Definidas por el Usuario y sus aplicaciones principales. También se analizan el paso de parámetros y la comprobación de los mismos y se estudia el uso del manejador de librerías LIB.EXE con el fin de crea nuestra propia biblioteca de funciones.

En el capítulo 6 se expone el manejo de los arreglos de N dimensiones, de algunas técnicas de inicialización y el uso de las funciones de Clipper que están relacionadas con los arreglos.

En el capítulo 7 "Code Blocks y macros" se explican las principales características de los Bloques de Código, algunas de sus técnicas de manejo y las ventajas sobre las macros.

En el capítulo 8 "La clase TBrowse y TBColumn" se explican algunas de las técnicas de manejo, y beneficios del uso de estas dos clases. Las cuales favorecen una programación modular, elegante y flexible para la manipulación de datos en forma de tablas.

El capítulo 9 describe el modo de crear aplicaciones en ambiente multiuasuario, analizando las funciones y comandos para trabajar en Red de Area Local (LAN) y los aspectos que deben considerarse para el desarrollo de este tipo de aplicaciones.

En el capítulo 10 "Interfase con Lenguaje C y Ensamblador" se explican las principales características del Sistema Extendido de Clipper.

1.1 CLIPPER COMO LENGUAJE DE DESARROLLO

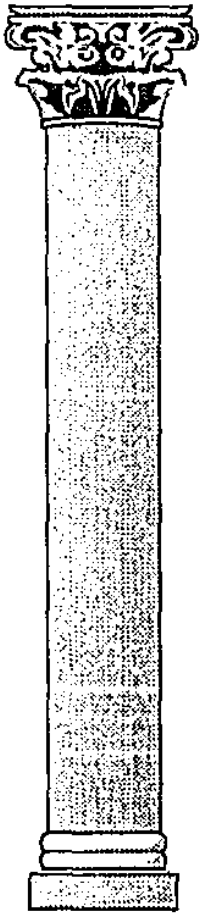
CLIPPER es un lenguaje de alto nivel, diseñado para desarrollo de sistemas Xbase. Es un compilador usado para el desarrollo de aplicaciones administrativas para computadoras personales. Tiene su origen en el lenguaje dBASE III Plus, ha extendido y mejorado el lenguaje de distintas maneras. Operaciones que típicamente requieren de muchos comandos en otro lenguaje de programación, pueden ejecutase simplemente en CLIPPER con un solo comando o función.

Por ejemplo, en CLIPPER y en general en sistemas Xbase no se tiene que declarar la estructura física del archivo de datos, como sí tiene que hacerlo en un lenguaje de programación de propósito general. Solo tiene que abrir el archivo de base de datos y se asignarán automáticamente buffers para el almacenamiento y estarán disponibles los contenidos de los campos. Esto requiere mucho mas esfuerzo si se usa un lenguaje de propósito general.

En Clipper, se requiere la programación de aplicaciones que serán creadas por el programador y distribuidas a los usuarios, debido a que no tiene un prompt ó centro de control que provea una interfase interactiva con el usuario, como sucede en dBase ó en FoxPro. Una vez escrito el programa, se compila y encadena para obtener un archivo ejecutable (.EXE), con lo cual los usuarios finales no tienen necesidad de aprender el manejo de bases de datos ó comandos, únicamente conocer el programa, en el cual el programador tiene la obligación de dejarlo debidamente documentado y de fácil uso.

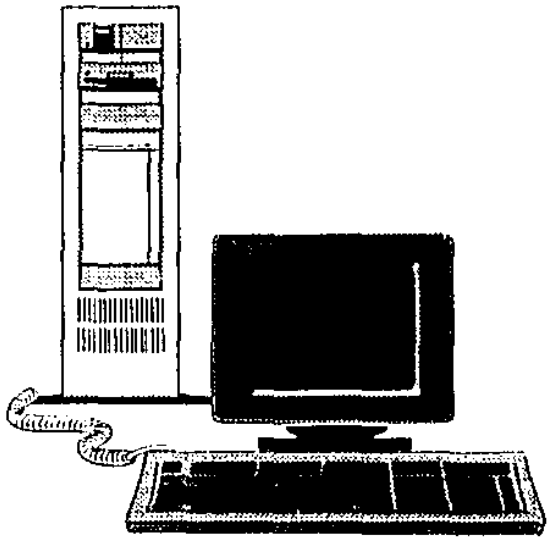
Los comandos y funciones de Clipper son una mejora del lenguaje del Dbase III+, se agregaron funciones y comandos más complejos y poderosos en alcance y tiempo de procesamiento basicamente y desaparecierón algunos de Dbase y algunos otros sufrierón cambios en su sintaxis y funcionamiento. En la versión 5.01 se incluyen muchas innovaciones. A continuación exponemos algunas, en los capítulos específicos de cada tema se ampliará la informacion:

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO



CAPITULO 1

- EL COMPILADOR Y SU AMBIENTE DE TRABAJO



ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES
A R A G O N

- Preprocesador que permite el uso de directivas de compilación para declarar constantes simbólicas, pseudo funciones, comandos definidos por el usuario, compilación condicional y archivos de cabecera.

- Nuevo enlazador (RTLINK) que aporta nuevas funcionalidades entre las que destacan los overlays dinámicos y las librerías preenlazadas.

- Array n_dimensiones y de dimensionamiento dinámico. Con CLIPPER 5.01 podemos pasar por referencia un array a una función y ésta así mismo, puede devolverlo.

- Programación orientada a objetos. Clipper 5.01 presenta varias clases de objetos programables: GET, TBROWSE, ERROR Y TBCOLUMN. Los objetos son elementos nuevos en la programación con CLIPPER. Cada uno de los cuatro determinan el modo de trabajo del entorno que configuran. Asociados a ellos existen una serie de variables de sistema, funciones, etc., cuyo uso sirve para determinar el modo en que trabajarán cada uno de ellos. Los valores dados a estas variables se envían a los objetos de forma que éstos se programen acorde con los mensajes recibidos.

- Nuevos modos LOCAL y STATIC para declarar variables. Estos se unen a los ya conocidos PRIVATE y PUBLIC.

- Posibilidad de tener funciones y procedimientos estáticos. Esto implica que un mismo nombre pueda emplearse para dos procedimientos distintos, siempre que se declare STATIC.

- Documentación en linea. CLIPPER 5.01 usa las excelentes NORTON GUIDES para tener en todo momento residente en memoria una completa ayuda a la programación.

- Nuevo y potente depurador permite, como una de sus posibilidades más interesantes, la ejecución del programa mientras se visualiza el código fuente del mismo.

- Incorporación del valor NIL (nulo).

- Un nuevo y amplio conjunto de variables de entorno que sirven para ajustar el funcionamiento óptimo de CLIPPER.
- Empleo configurable de la memoria expandida

- Code Blocks son un nuevo elemento de programación usado para hacer referencia a piezas completas de código compilado. Mediante los Code Blocks podemos resolver cuestiones tan interesantes como el paso de una función como parámetro a otra función o aun procedimiento.

Es importante cuestionarnos si estamos trabajando con la herramienta adecuada para el desarrollo de aplicaciones. ¿Es Clipper un producto estable?, ¿Es Clipper un producto con proyección?.

Nantucket Co. ha posicionado a Clipper entre los mejores productos en el mercado de lenguajes xBase y ha desarrollado una estrategia a la cual ha denominado NFT (Nantucket Future Technology), la cual pretende evitar sedentarismo de acuerdo a una estrategia que se resume a continuación:

a). Diseño Adaptivo de Plataformas

- Implantación de Multiplataformas
- Ejecución en Multiplataformas disimilares
- Mismo "Look & Feel" en todas las plataformas
- Portabilidad de archivos
- Ambientes de Interfase

b). Definición de Ambientes de Programación

- Ambientes que definan su propio "Look & Feel"
- Herramientas de Interfases que se comporten de la misma forma en cualquier ambiente.
- Manejo de elementos dependientes de la plataforma

c). Diseño independiente de la plataforma

- Manejo de información sin importar la plataforma
- Uso de abstracciones para generalizar servicios
- Soporte de dispositivos comunes a todas las plataformas (mouse, teclado, etc.)
- Soporte de elementos de interfases comunes (ventanas, menús, gráficas, etc.)

d). Diseño adaptativo de plataformas

- Comportamiento de aplicaciones, como si hubiésen sido desarrolladas para una plataforma específica.

1.2. REQUERIMIENTOS DE HARDWARE, SOFTWARE Y ESPECIFICACIONES TECNICAS

Clipper 5.01 puede funcionar en cualquier equipo IBM PS/2, AT, XT, SX, DX, PC ó 100% compatible que tenga las siguientes características:

1.2.1 Requerimientos De Hardware Y Software

- La capacidad mínima adecuada de memoria RAM debe ser 640 Kb (1 Mb de preferencia).
- Sistema operativo DOS versión 3.1 o superior (versión 3.3 en adelante de preferencia). Tanto para las aplicaciones monousuario, como multiusuario.
- Disco fijo con espacio libre de 6 Mb para instalar el producto. El disco duro es necesario para desarrollar aplicaciones aunque las aplicaciones desarrolladas pueden funcionar sobre disco flexible.

En caso de trabajo en ambiente de red de área local no se necesita un LAN PACK como sucede en el caso de dBase III+ y dBase IV. CLIPPER corre directamente sobre el sistema operativo de red sin necesidad de ningún producto adicional sin embargo, en cuanto a la cuestión de bloqueo de archivos y registros, ésta deberá realizarse siempre manualmente, contrariamente a lo que sucede con otros productos, donde el tratamiento de bloqueos se hace automáticamente. Compatible con sistemas operativos de red, como por ejemplo: Netware, LAN Manager, Vines, PC LAN, y 3+Open.

1.2.2 Especificaciones Técnicas

Número máximo de registros por base de datos	1000 millones
Número máximo de caracteres por registro	RAM disponible
Número máximo de campos por registro	1024
Número máximo de caracteres por campo	32
Dígitos máximos en un campo numérico	30
Campos memo (longitud variable)	64 Kb
Precisión en operaciones de cálculo	16 dígitos
Número máximo de caracteres en una clave de indexación	256
Número máximo de índices por área de trabajo	15
Número de variables de memoria públicas o privadas	2048
Número de variables de memoria locales o estáticas	RAM disponible
Tamaño máximo de una variable de memoria (cadena)	64 Kb
Número máximo de dígitos en una variable numérica	30 dígitos
Rango válido de fechas	1/1/100-1/12/9999
Número máximo de arrays	2848
Número máximo de elementos por dimensión de array	4096
Número máximo de dimensiones por array	RAM disponible
Número máximo de archivos abiertos (DOS 3.3 en adelante)	250

1.2.3 Características Generales

- Entorno Sistema Operativo DOS
- Espacio para la instalación completa 6 Mb
- Apoya Memoria Expandida
- Apoyo a la Memoria Expandida con caché de disco
- Compatible con Dbase III Plus
- Compatible con el formato de .dbf de Dbase III Plus
- Compatible con el formato de índice .ndx de Dbase III Plus (con manejador)
- Puede usar formato exclusivo de índice.

1.3 EL COMPILADOR

Cuando queremos comunicarnos con la computadora debemos hacerlo directa o indirectamente en su lenguaje. Para ello podemos escribir en código máquina, lo cual resulta lento y muy pesado o utilizar un lenguaje traductor que se encargue de hacer la conversión al lenguaje máquina, de tal forma que introducimos una serie de órdenes en un lenguaje similar al que utilizamos para comunicarnos con las personas y el traductor se encarga de traducirlo al lenguaje interpretable por la máquina.

1.3.1 Traductores

Los traductores son programas que permiten transformar o traducir una serie de caracteres comprensibles al ser humano (programa fuente), a otra serie de caracteres compresibles a la computadora o al microprocesador (programa objeto).

Los traductores toman como entrada un programa desarrollado en un lenguaje pseudo-natural, y genera un programa objeto con instrucciones fundamentales para el procesador (lenguaje máquina).

Existen dos grupos de traductores:

- Intérpretes
- Compiladores

Los interpretes son aquellos programas que analizan el programa fuente línea por línea, y la transforman al microprocesador para que de forma inmediata ejecute el código que se desea. Es decir, analiza un estatuto, generan el o los tokens correspondientes y lo ejecutan, continuando con este proceso hasta que termina el programa fuente , o bien existe un error.

Los intérpretes tienen ventajas importantes, por ejemplo, al realizar un cambio en el programa fuente, se puede observar su ejecución de forma inmediata, debido a que la ejecución se va efectuando línea por línea. Por otro lado, una de las grandes desventajas consiste en la lentitud de ejecución del código.

Pese a las desventajas que tiene un intérprete, en ocasiones se convierte en una herramienta sumamente útil precisamente por la ventaja de poder observar un cambio en el programa fuente de forma inmediata.

Los compiladores son aquellos programas que analizan un programa fuente, línea por línea hasta generar un programa objeto. Durante el proceso de traducción, un compilador puede identificar errores en el programa fuente, y a diferencia de los intérpretes, no ejecutan el código analizado.

Existen dos parte en el proceso de compilación: Análisis y Síntesis. La etapa de Análisis descompone el programa fuente en unidades (llamadas tokens) y genera una representación intermedia del código. La etapa de Síntesis construye el programa objeto tomando como base la representación intermedia.

Adicionalmente al compilador, se requiere de otros programas para generar código ejecutable. Un programa fuente puede estar separado en módulos en diferentes archivos. El proceso de recolectar los módulos se

efectúa a través de un programa llamado Preprocesador, el cual expande macros.

Etapas del proceso análisis en un compilador:

1). Análisis de Léxico: Generador de tokens.
2). Análisis de Sintaxis: Generador de código intermedio.
3). Análisis de Semántica: Verificador de tipos.

Etapas de un proceso de síntesis de compilación:

1). Preprocesador: Generador de código fuente.
2). Compilador: Generador de código objeto (.obj)
3). Encadenador: Generador de código ejecutable (.exe).

Las ventajas de un compilador se enfocan principalmente a la velocidad de ejecución del código, así como el manejo de la administración de memoria además; el programa fuente queda protegido; mientras que una de sus principales desventajas es el tiempo que se consume en las etapas de este mismo proceso para la generación de código ejecutable.

1.3.2 Opciones Del Compilador

Clipper cuenta con una serie de opciones o switches para compilar un programa de acuerdo a las necesidades específicas. La sintaxis a emplear es:

CLIPPER [Aprg | @<Fclp>] [/<opción> {/<opción>}]

donde <Aprg> es el nombre del archivo (.prg) .Las opciones pueden definirse en la variable de ambiente, o bien tecleándolas directamente. Dichas opciones son las siguientes:

/A Declaración automática de variables públicas y privadas.

Esta opción permite definir los identificadores públicos y privados

como variables de memoria. Este switch es importante si se utilizan variables públicas, debido a que le evitarán errores de referencia ambigua.

/B Prepara programa para Debugger.

Debido a la información que requiere la nueva versión del debugger, es importante indicarle a Clipper que añada al código objeto como número de linea, nombres de variables y de archivos. Si se utiliza esta opción, evite el uso del switch /L.

/D Define un identificador.

Esta opción permite definir un identificador al procesador. La sintaxis a utilizar es:
/D<identificador> [= <texto>].

/I Expande la búsqueda de archivos de cabecera (hearder files).

Cuando requerimos que Clipper busque un "header file" en un directorio diferente al especificado en la variable de ambiente INCLUDE, entonces podemos utilizar esta opción como: /I<ruta>.

/L Suprime número de línea.

Cuando se compila un programa sin utilizar este switch, el código objeto almacena 3 byte por cada línea del programa fuente. Una vez que su aplicación esté libre de error, utilice esta opción para reducir el código.

/M Compila el módulo actual.

Esta opción ignora referencias externas del programa compilado, y es útil sobre todo si pensamos en el hecho de la programación modular. Por ejemplo, si en el programa a compilar se hace referencia a funciones y/o procedimientos, Clipper los compilará junto con el actual, dejando un solo archivo objeto.

/N Suprime procedimiento principal.

Posibilita que un procedimiento o función interna a un programa (.prg) pueda tener el mismo nombre que este último.
Cuando Clipper compila un programa, genera un procedimiento con el mismo nombre que el archivo .Prg, de tal forma que cuando se ejecute, se referenciará a dicho nombre.

Si se utiliza esta opción, el primer procedimiento o función que se ejecutará será la primera que el linker encuentre al momento de resolver las referencias.

/O Genera un archivo (.obj) al nombre especificado.

Cuando se utiliza esta opción, Clipper generará el programa objeto con el nombre especificado. Si únicamente utilizamos una ruta, ésta debe terminar con el carácter "\". La sintaxis para utilizar esta opción: /O<nombre>

/P Evita que el compilador borre el archivo del preprocesador.

Cuando compilamos un programa, la primera etapa de éste proceso es el preprocesador, el cual genera código fuente (.PPO). Dicho código fuente generado por el preprocesador es eliminado. Con este switch, evitamos que el archivo generado sea eliminado.

/Q Suprime mensajes de líneas compiladas.

Este switch únicamente sirve para evitar que Clipper despliegue el número de líneas compiladas.

/R Indica al encadenador donde buscar símbolos no resueltos.

Para indicar al encadenador en que librerías debe buscar aquellas funciones que no pertenecen a 5.01, podemos utilizar éste switch como: /R<nombre de librería>.

S/ Verifica sintaxis.

Cuando se utiliza éste switch, Clipper no genera código objeto; sin embargo si analiza la sintaxis del programa fuente.

/T Especifica la ruta para archivos temporales.

Durante el proceso de compilación, Clipper genera archivos temporales que al final del proceso son eliminados. Utilizando éste switch podemos indicarle una ruta en donde los archivos temporales serán creados. La sintaxis de éste switch es: /T<ruta>.

/V Trata las referencias ambiguas como dinámicas.

Esta opción permite definirle a Clipper que todas aquellas variables no declaradas, sean tratadas como variables de memoria. Utilice esta opción para compilar código de Summer'87.

/W Genera mensajes de advertencia.

Esta opción genera mensajes de advertencia para variables que no han sido declaradas. Utilice este switch para comenzar la conversión de código de summer'87 a 5.01.

Si compila código de sumer'87, puede utilizar los siguientes switches:

CLIPPER <Aprg> /m/l/v

Para compilar código de Clipper 5.01 es conveniente utilizar los siguientes switches:

CLIPPER <APrg> /n/m/w

y opcionalmente

CLIPPER <Aprg> /n/m/w/p/b

1.4 VARIABLES DE AMBIENTE

Clipper usa dos métodos para controlar el ambiente DOS: Los archivos de arranque y las variables de ambiente DOS.
Cuando una PC es inicializada, dos archivos (autoexec.bat y config.sys) establecen las condiciones del sistema, de acuerdo a los recursos y a la conveniencia del usuario. Clipper utiliza ambos archivos para establecer su ambiente.
El segundo método para controlar el DOS es usar el comando SET en el apuntador del DOS para crear variables en el ambiente DOS.

- Controlando DOS con archivos de arranque

  Al menos, el Config.sys debe contener dos líneas: La referente a los archivos y la referente a las memorias buffer, por ejemplo:

  files =20
  buffers=8

  Clipper recomienda 8 buffers, pero si algún otro paquete que utilizamos requiere de más, entonces debemos permitir que el otro establezca cuantos.

  Después de que son cargadas las opciones del config.sys, se ejecuta el autoexec.bat. Este archivo es usado para ejecutar comandos DOS cada vez que se enciende la computadora. Se puede usar el autoexec.bat para establecer ciertas condiciones en el ambiente DOS, que usarán los programas compilados en CLIPPER.

- Controlando el DOS con variables.

  El ambiente DOS es una espacio en memoria RAM en donde el DOS y los programas pueden asignar valores a variables que serán leídas conforme se vayan necesitando. Si queremos ver está información en nuestro sistema, basta con teclear SET desde el apuntador del DOS y todos los valores actuales serán desplegados. Sintaxis para crear una nueva variable:

SET <variable> = <expresión>

por ejemplo:

SET VAR=TESIS

Cuando se ejecute el comando SET, se podrá observar a VAR en la lista de variables y su valor será la cadena TESIS. Es importante no dejar espacios en blanco antes y después del signo igual.

El tamaño por default del ambiente DOS es de solo 160 bytes. Debido a que Clipper usa el ambiente extensamente, es posible que se deba incrementar dicho tamaño en algunas ocasiones. Esto se hace colocando la siguiente línea en el config.sys:

SHELL=C:\COMMAND.COM E: <numbytes> /P

donde <numbytes> es la cantidad deseada en bytes para el ambiente. el rango esté entre 160 y 32768. Generalmente 1034 son suficientes. Para versiones de DOS 3.1 y anteriores, el argumento E especifica el número de párrafos de 16 bits de memoria asignados por el ambiente.

Las variables de ambiente son necesarias para CLIPPER de tal forma que pueda operar adecuadamente. Dichas variables son localidades de memoria que se establecen a través del Sistema Operativo, y que por medio de las cuales se puede configurar el entorno de la PC.

El espacio asignado a las variables de ambiente es fijo, sin embargo puede modificarse esta asignación siempre y cuando cuente con DOS 3.2 en adelante.

Variable del Compilador: Es conveniente definir en la variable de ruta (path) el directorio en donde se encuentra Clipper.exe para poder referenciarlo. Por ejemplo:

SET PATH=C:\CLIPPPER5\BIN

Variable de Inclusión:

La variable de inclusión se refiere a la ruta que deba tener dicha variable para poder incluir los archivos de encabezado (header files), al momento de compilar. Esta variable tiene un nombre específico, ejemplo:
SET PATH=C:\CLIPPER5\INCLUDE

Variable de librería:

La variable de ambiente de librería se refiere a la ruta que deba tener para efectos de que el proceso de encadenamiento, el linker pueda encontrar las librerías necesarias para resolver símbolos. Esta variable tiene un nombre específico, por ejemplo:
SET LIB=C:\CLIPPER\LIB

Variable de objeto:

Esta variable de ambiente se utiliza para que clipper almacene temporalmente programas objeto que produce durante la compilación. Tiene un nombre específico y puede definirse como:

SET OBJ=C:\CLIPPER5\OBJ

Variable de librerías Pre-encadenadas:

Esta variable permite definir la ruta del directorio en dónde se localizan las librerías preencadenadas que se utilizan en el momento de la ejecución. Por ejemplo:

SET PLL=C:\CLIPPER5\PLL

Variable de temporales:

Esta variable se utiliza para indicarle a Clipper en dónde debe crear los archivos temporales. Por ejemplo:

SET TMP=C:\CLIPPER5\TEMP

Variable de switch: La variable de ambiente de switch se utiliza para efectos de compilar con opciones deseadas. Por ejemplo:

SETCLIPPERCDM=/M/N/W

Finalmente, se puede hacer uso de la variable de ambiente SET CLIPPER para configurar efectos de ejecución.

Clipper usa el ambiente de diferentes maneras:

a). Puesto que Clipper es cargado en memoria, usa la variable SETCLIPPER para determinar el número de factores ambientales.

b). Clipper usa la función GETENV() para que las aplicaciones puedan leer variables de ambiente DOS.

c). Clipper usa variables de ambiente para localizar archivos y establecer defaults para el compilador y el encadenador.

Cuando se carga una aplicación Clipper en memoria, se busca en el ambiente DOS si la variable SET CLIPPER, si ha sido establecida. Si existe, la variable CLIPPER es usada para controlar 8 diferentes opciones. Si mas de una es seleccionada, las opciones estarán separadas por punto y coma (;).

En la versión 5.01, varias de las opciones SET CLIPPER son diferentes y algunas de la versión SUMMER 87 no son soportadas. La sintaxis es:

```
SET CLIPPER= [E<NumKbExpandida>]
             [;F<NumManejadores>]
             [;X<NumKb>]
             [;DYNK:<NumKb>]
             [;DYNF:<NumManejadores>]
             [;SWAPK:<NumKb>]
             [;SWAPPATH <"UnidadDirectorioArchivo">]
             [;CGACURS]
             [;BADCACHE]
```

[;EXTHEAP:<NumKb>]

E<NumKbExpandida>:

La opción E limita la cantidad de memoria expandida que usará la aplicación. La memoria expandida no es asignada dinámicamente. Normalmente, toda la RAM expandida disponible en el sistema (hasta 8 Mb) es asignada por el Manejador de Memoria Virtual (VMM). Agregando esta opción en SET CLIPPER, se le indica al manejador de memoria el máximo de Kb de memoria expandida que puede usar.

F<NumManejadores>:

Las versiones de DOS 3.2 y menores tienen el límite de 20 manejadores de archivos. Debido a que DOS toma cinco manejadores para sí mismo, las aplicaciones en Clipper están limitadas a 15 archivos abiertos. Desde DOS 3.3 en adelante permite hasta 255 manejadores en la sentencia files del config.sys. Por lo tanto, las aplicaciones de Clipper pueden tener hasta 250 archivos abiertos. Para aquellas aplicaciones de Clipper que requieran más de 15 archivos abiertos, nos debemos asegurar que la sentencia files del config.sys tenga los manejadores suficientes para la aplicación, después usar la opción F de la variable SET CLIPPER. Ambas condiciones indicarán que mas de 15 archivos pueden ser abiertos. La condición SET CLIPPER no puede ser mayor a la especificada en el config.sys.

X<NumKb>:

La opción X es usada para indicarle a la aplicación hecha en Clipper que ignore una porción de la memoria convencional. Esto sirve para examinar como se comportaría la aplicación en un sistema que tuviera menos memoria. Sea cuidadoso con el comando RUN/!, por que DOS no respetará la porción indicada por la opción X.

DYNK:<NumKb>:

Esta opción controla la cantidad de memoria que es asignada para el Sistema Dinámico de Overlay (Dinamic Overlay System). Por default, una aplicación Clipper asigna un porcentaje de la memoria para el área de overlay dinámico. Si se pone erróneamente esta opción, puede afectar severamente la eficiencia de la aplicación. Nantucket no recomienda el uso de esta aplicación.

DYNF:<NumManejadores>:

Esta opción controla cuantos manejadores de archivos puede usar el Sistema Dinámico de Overlay. El default es 2.
SWAPK:<NumKb>:
Esta opción limita el tamaño del archivo de intercambio usado por el administrador de memoria virtual. El Disk File de intercambio es usado solo después de que toda la memoria es agotada.

SWAPPATH <"UnidadDirectorioArchivo">:

Esta opción es usada para especificar un path donde el archivo de intercambio es escrito por el VMM (Manejador de Memoria Virtual). Si no se especifica, es usado el directorio actual.

CGACURS:

Esta opción puede ser usada en máquinas viejas que no soportan las características extendidas EGA/VGA del cursor. Esta opción puede ser usada para evitar problemas cuando las aplicaciones Clipper corren en ambiente multitarea o cuando programas residentes en memoria son cargados.

BADCACHE:

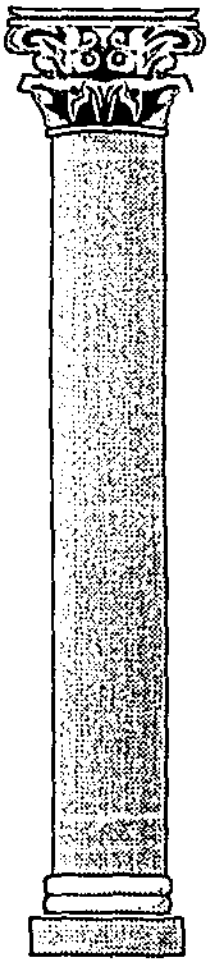
Este argumento es usado para prevenir conflictos cuando trabaja mal un programa disco caché. Esto ocurre cuando el software caché no espera que algún otro software use RAM expandida. Cuando se proporciona este argumento a la variable CLIPPER, el VMM se estabiliza así mismo

cada vez.

EXTHEAP:<NumKb>:

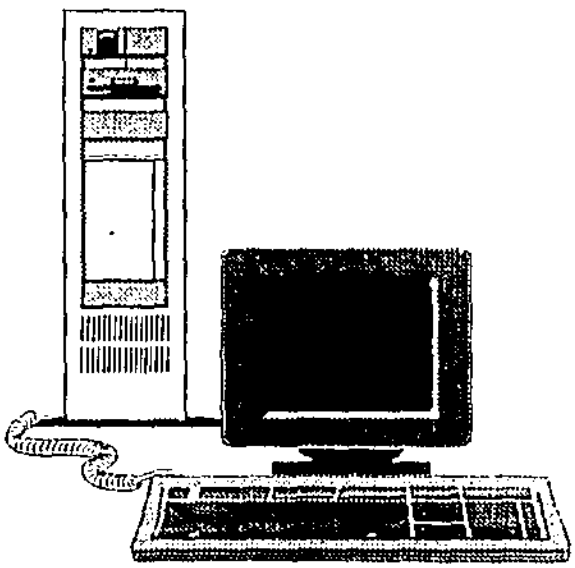
Por default Clipper reserva 8 Kb de memoria para ser usada por funciones escritas en lenguaje C que necesitan localidades extra de memoria. Normalmente esa cantidad es suficiente, pero una función extremadamente grande podría requerir mas. En tal caso la memoria de uso puede ser fragmentada en cuanto el VMM haga una petición de memoria disponible. Usando el argumento EXTHEAP en la variable SET CLIPPER, se pueden acomodar las funciones de C sin fragmentación de memoria.

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO



CAPITULO 2

- EL ENCADENADOR RTLINK, LIBRERIAS Y HERRAMIENTAS MAKE



ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES
A R A G O N

2.1 EL ENCADENADOR RTLINK

El último paso para la creación de un programa ejecutable consiste en encadenar o enlazar todos los simbolos (referencias e identificadores). Este proceso se ejecuta a través de un programa encadenador o linker.

El objetivo de un encadenador es el de asociar los módulos objetos, obtenidos mediante el compilador, con las librerías donde se contienen las traducciones máquina de cada una de las sentencias que aparecen en el modulo objeto. El producto proporcionado por el encadenador es un módulo ejecutable (.EXE) que es el que ya podemos hacer funcionar invocando simplemente su nombre.

El encadenador que proporciona CLIPPER 5.01 es RTLINK, el cual es un programa complejo en términos de lo que puede hacer y en la forma en que se puede utilizar.

2.1.1 Que es el encadenador RTLINK

Una vez que se captura el código de la aplicación, lo que se tiene es un archivo fuente (en código ASCII) con extensión .PRG. Posteriormente al compilarse, lo que se obtiene es un programa objeto.

Pero todavía este programa no es ejecutable, ya que tiene que ser enlazado con las librerías propias del programa compilador. El enlace necesario lo realiza un programa encadenador llamado RTLINK. Si los archivos objeto no son enlazados, no podremos hacer nada con ellos, puesto que ni están en ASCII y ni son ejecutables. Son unos archivos intermedios entre el código fuente de la aplicación y el archivo ejecutable.

El programa ejecutable es fruto de haber enlazado los módulos objeto con las librerías adecuadas, dando como resultado un archivo .EXE.

2.1.2 Sintaxis y Llaves RTLINK

Sintaxis para usar el encadenador:

RTLINK [FILE <lista de objetos> [OUTPUT <arch_salida>] [LIBRARY [<lista de librerías>] [<lista de opciones>]] | @Flnk]

LLAVES	ABREVIATURAS
FILE	FI
OUTPUT	OU
LIBRARY	LIB
BATCH	BAT
NOBATCH	NOBAT
IGNORECASE	IG
NOIGNORECASE	NOIG
DYNAMIC	DYN
RESIDENT	RES
DEBUG	DEB
HELP	H
MAP	MAP
SILENT	SI
VERBOSE	VE
PRE-LINK	PR
PLL	PL
INCREMENTAL	INC
NOINCREMENTAL	NOI

Tabla 2.1 Las llaves de Rtlink más usadas

RTLINK utiliza llaves para indicar que opciones se usarán en el encadenado. Las llaves pueden ser especificadas en cualquier orden y son válidas en cualquiera de los modos de operación: comandos en línea, en forma interactiva y en archivo Script. La llave puede ser tecleada en su forma completa ó usar las abreviaturas. La tabla 2.1 es una lista parcial de las llaves mas comúnmente usadas y se da una

breve descripción de su uso:

/FI <arch1> [<,arch2>]

Denota una serie de archivos objeto separados por comas, para ser encadenados.

/OU <arch>

Especifica el nombre y ruta final del archivo ejecutable cuando deseamos que sea diferente al del primer archivo objeto relacionado.

/BAT | NOBAT

Indica de que modo se portará el encadenador cuando no encuentre los archivos necesarios para el encadenamiento. En el modo NOBAT (default) se desplegarán mensajes en la pantalla, preguntando en donde puede encontrar un archivo objeto, una librería o una librería preecadenada. En el modo BAT no se hacen preguntas. En lugar de eso presenta un mensaje de error y aborta la operación, poniendo el valor de ERRORLEVEL en 1.

/IG | NOIG

En el modo IG (default) se tomaran por igual las mayúsculas y minúsculas, no siendo así en el modo NOIG.

/DI [INTO <arch>] | RES

Establece si una rutina se guardará en memoria dinámica o residente. El default es memoria dinámica excepto para C y ensamblador. La opción INTO <arch over> sirve para poner parte de código ejecutable

en un archivo overlay con extensión .OVL. Con la opción RES la rutina se guardará en memoria residente.

/DEB

Causa que la aplicación despliegue un mensaje, identificando a los archivos overlay cuando son cargados en memoria.

/H

Proporciona una lista completa de todas las llaves disponibles para RTLINK.

/MA [=<map_arch>] [<map_opciones>] [S] [N] [A]

Esta opción hace que se cree un archivo conteniendo uno o mas reportes de memoria que representen el conducto por el cual una aplicación es puesta en memoria. Existen tres diferentes mapas disponibles:

S	Segmentos con direcciones de segmentos
N	Símbolos PUBLIC ordenados por el nombre del símbolo
A	Símbolos PUBLIC ordenados por dirección

Si no se especifica uno o mas reportes, el encadenador creará a todos ellos en un archivo.

/SI | /VE [<Nivel N>]

La opción VE despliega la información en pantalla de lo que esta haciendo RTLINK mientras ejecuta la operación de enlace. Y además el nivel de mensajes que se despliegan. SI suprime el despliegue de

mensajes y apuntadores y es similar al switch QUIET del compilador.

/PR

Le indica al encadenador que se creará una librería preencadenada en lugar de un archivo .EXE. Además se crea un archivo .PLT que contiene la información actual del contenido de dicha librería.

/PLL <NombPLL>

Se indica al encadenador que el archivo .EXE que se creará necesitará de una librería preencadenada de nombre <NombPLL>. Lo que significa que no estará incluido todo el código en el archivo ejecutable, por lo que necesitará la librería preencadenada.

/NOI | INC [PorcenDesperd]

Esta opción indica de que modo se hará el encadenamiento. El modo por default es NOINC, el cual no deja espacio entre los módulos encadenados. El modo INC deja un espacio [PorcenDesperd] entre cada módulo que va encadenando y solo re enlaza aquellos módulos que han sufrido cambios, basándose en la fecha y hora del archivo. El porcentaje por default es 25%. Además crea un archivo .INF el cual informa los cambios efectuados en el último encadenamiento.

2.1.3 Como se invoca a RTLINK

Existen 3 formas de invocar a RTLINK. A continuación se dará una breve explicación de cada método.

a) RTLINK en Línea.- La primera forma es poner en línea todas las opciones y direcciones. Por ejemplo, en el apuntador del DOS

podríamos teclear algo como lo siguiente:
RTLINK /VE FI miprogra

b) RTLINK en modo Interactivo.- Si simplemente tecleamos RTLINK y presionamos <Enter> desde el DOS, se habilitará la entrada de opciones y direcciones línea por línea. Este modo interactivo continua hasta que se teclee punto y coma (;) y <Enter> o se presione Ctrl-Z y <Enter>. En modo interactivo, el ejemplo anterior se vería así:

```
RTLINK
=>VE
=>FI miprogra
=>;
```

Después de teclear <;> presione <Enter> y el encadenador producirá el archivo .EXE.

c) RTLINK mediante un archivo Script.- Se puede controlar el proceso de encadenado mediante un archivo en código ASCII al que se le conoce como Script. Su extensión por default es .LNK. En este archivo se guardan todas las opciones y direcciones deseadas. Para usar este método se teclea en el apuntador del DOS un comando con la siguiente sintaxis:

RTLINK @<miscript>

Donde <miscript> es el nombre del archivo Script. El contenido de este archivo para el ejemplo anterior se vería así:

RTLINK /VE FI miprogra

2.1.4 Como se configura a RTLINK

Existen cuatro formas para indicarle a RTLINK que opciones se quieren usar en el encadenamiento de la aplicación. Cada una tiene un orden de procedencia. De esta manera se pueden invalidar los defaults del RTLINK y establecer los que se deseen. A continuación se dará una breve explicación de cada método. Empezando con el nivel mas bajo de prioridad, estos son los cuatro métodos:

a) El archivo de configuración RTLINK.CFG.- Este es el nivel de control mas bajo en prioridad. Cuando RTLINK es invocado, este busca en el subdirectorio actual al archivo RTLINK.CFG. Sí no lo encuentra, RTLINK busca en el subdirectorio en donde se encuentra el RTLINK.EXE (generalmente c:\clipper5\bin). Sí aquí tampoco es encontrado, la búsqueda se abandona y se asume que el archivo CFG no existe. Este tipo de archivo está en código ASCII y puede ser creado y editado.

b) La variable de ambiente DOS: RTLINKCMD.- El encadenador usa una variable de ambiente llamada RTLINKCMD para almacenar valores por default. Por ejemplo, se puede establecer que por default se informe de lo que está sucediendo durante el encadenamiento; para esto, agregue el siguiente renglón en el archivo autoexec.bat

   SET RTLINKCMD=/VE

   En caso de haber conflicto, cualquier cosa establecida en este comando tiene prioridad sobre lo establecido en el archivo .CFG.

c) Incluir opciones en la línea de comandos.- Es una forma que permite establecer los valores por default en forma interactiva. Este método tiene prioridad sobre las dos opciones anteriores. (Se dio un ejemplo de este método en el punto 2.1.3).

d) Archivos Script encadenadores (.LNK).- El nivel más alto de prioridad para establecer condiciones es el proporcionado por un archivo script. La extensión por default para un archivo script es .LNK. Para llamar al encadenador con un archivo script, deberá teclear algo como lo siguiente:

RTLINK @<miscript>
Donde <miscript> es el archivo Script encadenador, con extensión .LNK. Si hay conflicto en lo establecido en este archivo con respecto a los métodos anteriores, se resuelve a favor del archivo Script.

Los archivos .LNK se crean con un editor ASCII. El contenido de los mismos ha de estar compuesto únicamente por comandos y opciones del enlazador: FILE, LIB, etc.

2.1.5 RTLINK y el DOS ERRORLEVEL

Cuando el RTLINK completa exitosamente el proceso de encadenado, (sin errores), el DOS ERRORLEVEL es puesto a 0. Si un error ocurre, o el encadenamiento es terminado presionando Ctrl-C en el teclado, el DOS ERRORLEVEL es puesto a 1. Los errores comunes incluyen errores de sintaxis en los comandos en línea, errores en el archivo script, y la incapacidad del encadenador para encontrar los archivos necesarios. Cuando ocurre un error, el código no es generado y un mensaje de error es desplegado.

2.1.6 Overlays de Clipper 5.01

Cuando se está desarrollando una aplicación, es imposible conocer la magnitud del tamaño del sistema, en lo que se refiere a la capacidad de la memoria que se requerirá para que el desarrollo funcione adecuadamente.

Desafortunadamente no existe algún método para conocer de antemano el tamaño real del programa (haciendo referencia a la memoria), y hasta que probamos la aplicación en su totalidad, súbitamente nos encontramos con "OUT OF MEMORY". La solución radica en el diseño de overlays.

El realizar overlays es una técnica que permite ejecutar un programa en un segmento pequeño de memoria, de esta forma puede realizar sistemas extremadamente grandes.
Un overlay es un segmento de la aplicación que no está en memoria RAM hasta que la ejecución de éste sea requerida. El código de su programa principal (raíz) permanece en memoria, mientras que los segmentos permanecen en disco. La parte de la memoria que estos overlays comparten es llamada área de overlay. Cuando se carga a memoria un segmento de la aplicación ocupa esta área, cuando se llama a otro segmento diferente de la aplicación, éste se incrustará en la misma área sobreponiéndose al segmento anterior.

El tamaño de ésta área será tan grande como el requerimiento de memoria del overlay más grande. Un programa que contiene un solo overlay, no reduce el tamaño de la memoria necesaria para la ejecución. La eficiencia de la memoria consiste en que varios overlays ocupen en forma alternativa el mismo espacio.

Clipper 5.01 puede manejar tanto overlay estáticos como dinámicos. Además pueden ser internos o encadenados como archivos externos, con extensión .OVL. La característica fundamental de RTLINK consiste en la habilidad que tiene para crear automáticamente overlays dinámicos internos.

Los overlays dinámicos de la versión 5.01 hacen que el programador se olvide de las dimensiones que tendrá su aplicación. El enlazador se encarga de preparar el programa ejecutable para que vaya cargándose en memoria RAM según las necesidades del sistema.

A pesar de la mejora reseñada, Clipper 5.01 sigue permitiendo el uso de overlays estáticos. Esto presenta como ventaja el hecho de que es el programador quien decide que partes del código cargar y descargar en la memoria, según las necesidades de espacio que se generen.

Se recomienda que aproveche todas las ventajas de los overlays dinámicos, pero sin olvidarse de los estáticos.

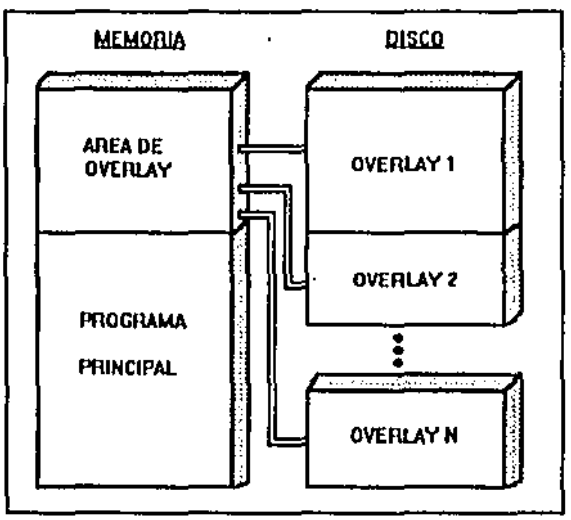


Figura 2.1 Forma en la que los overlay se cargan a memoria. Nótese que el tamaño del área del overlay corresponde al del overlay mas grande.

Para poder comprender como se crean los overlays vamos a suponer que tenemos una aplicación que posee la siguiente estructura de módulos (Figura 2.2):

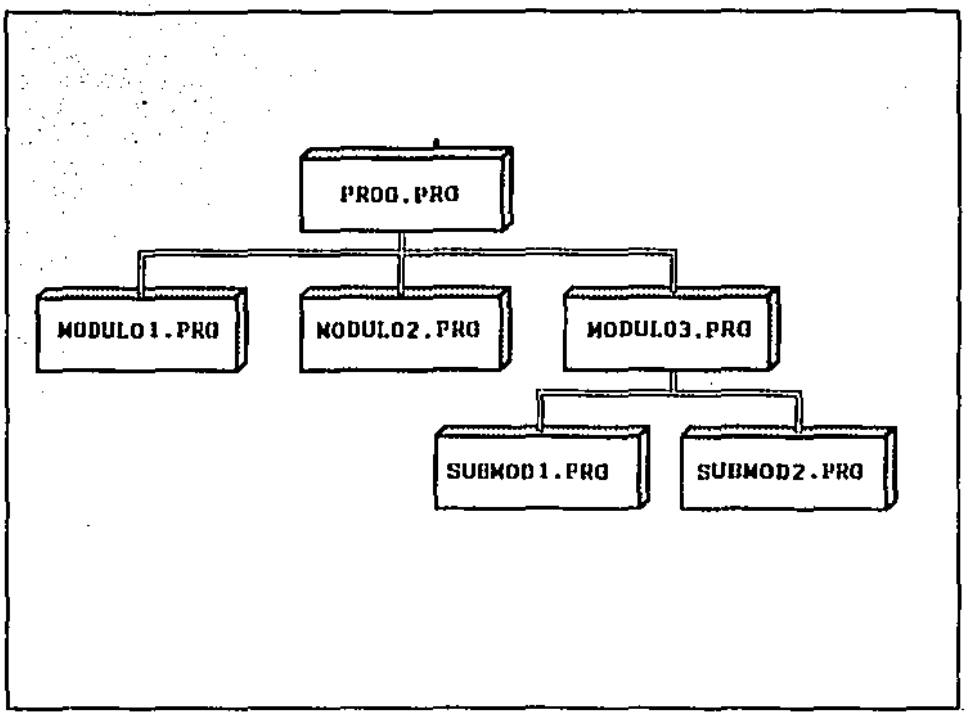


Figura 2.2 Estructura del programa Prog.prg

Dependiendo del tamaño de los módulos podemos emplear las siguientes técnicas.

OVERLAYS ESTATICOS

Son aquellos que contienen rutinas especificadas por el programador, así como el área para ser utilizadas por dichas rutinas. Este espacio de memoria es reservada al momento de encadenamiento y tiene un tamaño fijo. Todas las rutinas en lenguaje C o ensamblador deberán ser siempre estáticas.

Los overlay estáticos permiten diversas técnicas:

TECNICA 1

Si el ejecutable ocupa demasiado espacio para caber en la memoria y no queremos usar la técnica estándard de overlays dinámicos, podemos proceder a hacer una división sencilla en overlays estáticos. Situaremos un área principal donde estará prog.prg y una área de overlay donde se irán cargando los otros módulos según sean llamados.

Las ordenes necesarias para crear estos overlays, podemos introducirlas en un archivo script al que denominaremos encadena.lnk. El contenido del mismo debería ser:

```
FILE prog
BEGINAREA
  SECTION FILE modulo1
  SECTION FILE modulo2
  SECTION FILE modulo3
ENDAREA
```

Los mandatos necesarios para proceder al compilado y encadenado de esta aplicación serían:

```
CLIPPER prog -m
CLIPPER modulo1
CLIPPER modulo2
CLIPPER modulo3
RTLINK @encadena
```

La creación de un área de overlay estático la producen los mandatos BEGINAREA y ENDAREA. Los distintos mandatos SECTION FILE sirven para crear cada uno de los módulos que se cargarán en el área definida.

El parámetro -m hace que no se compilen las llamadas DO realizadas en este archivo. Esto hará que el archivo prog.obj sólo contenga el código necesario para su propia ejecución sin tener en cuenta al resto

de los módulos a los que llame. Estos son compilados por separado. En ellos no se indica ya la opción -m, así en modulo3.prg, las llamadas DO submod1 y DO submod2, harán que el código de esos archivos fuente se incorpore el módulo objeto modulo3.obj.
RTLINK @encadena une los cuatro módulos objeto que se han obtenido tras las instrucciones de compilación El nombre del módulo ejecutable es del primer objeto relacionado, es decir, prog.exe.

Visualmente la memoria quedaría del siguiente modo (Figura 2.3):

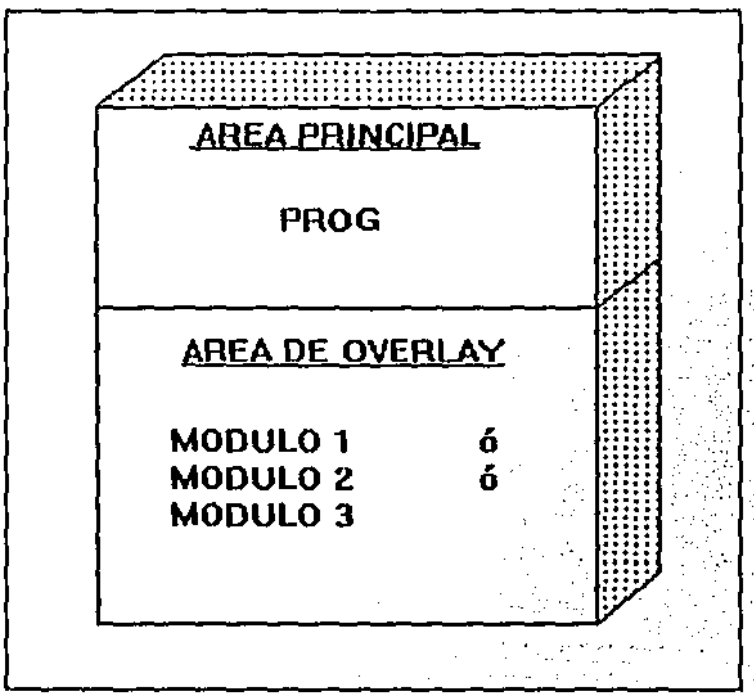


Figura 2.3

TECNICA 2

También se puede crear más de un área de segmentos para incorporar los diversos módulos de una aplicación. Así en el ejemplo anterior podríamos cambiar las sentencias de encadena.lnk, dejando:

```
FILE prog
BEGINAREA
 SECTION FILE modulo1
 SECTION FILE modulo2
ENDAREA
BEGINAREA
 SECTION FILE modulo3
ENDAREA
```

Las sentencias de compilación necesarias son las mismas que en el ejemplo anterior. La diferencia esta en que esta aplicación, la memoria quedaría del siguiente modo (Figura 2.4):

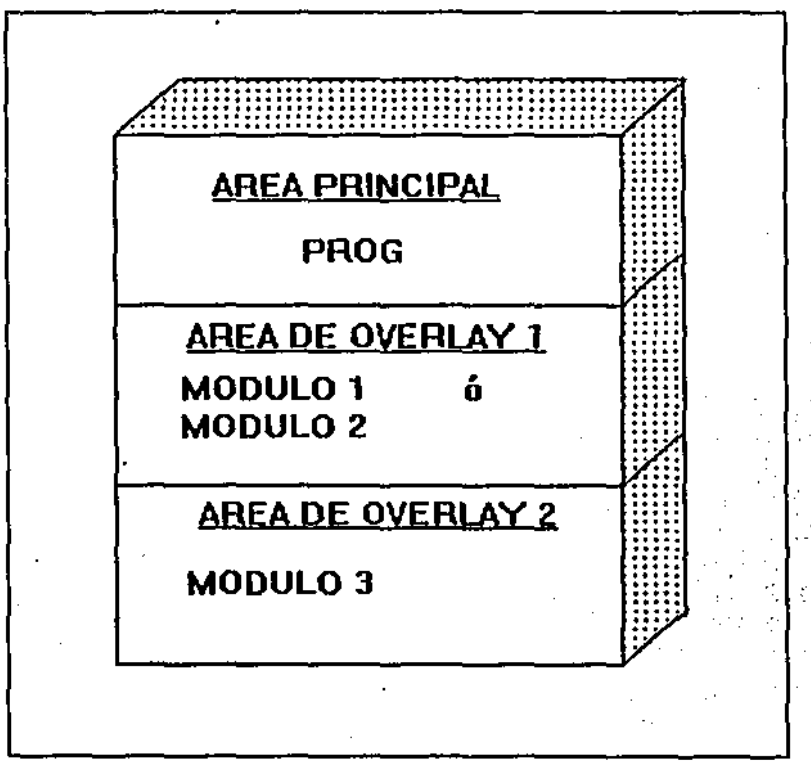


Figura 2.4

Así, en la memoria de nuestro ordenador conviven siempre tres áreas. Una de ellas, la principal, contiene permanentemente el código de prog. En cada una de las dos áreas de overlays se van cargando, conforme

son llamados, cada uno del resto de los módulos. En este caso la ventaja de dejar un área independiente para modulo3 puede partir de que este programa, como mas arriba hemos indicado, se supone que es el mas extenso de los tres, ya que contiene dos submódulos submod1 y submod2, Esta técnica suele requerir para su uso más cantidad de memoria que la anterior.

TECNICA 3

Los overlays estáticos también pueden anidarse. Es decir, un determinado módulo puede contener determinados submódulos que se cargan al ser requeridos desde él. Esto supone que, ya no en una área de segmentación, sino un módulo de los que se cargan en dicha área, puede ser subdividido. Esta técnica requiere una planeación muy cuidadosa de nuestros programas. Nuestro ejemplo podría tener una anidación en modulo3. El código necesario para el archivo encadena.lnk sería:

```
FILE prog
BEGINAREA
  SECTION FILE modulo1
  SECTION FILE modulo2
  SECTION FILE modulo3
  BEGINAREA
    SECTION FILE submod1
    SECTION FILE submod2
    SECTION FILE submod3
  ENDAREA
ENDAREA
```

Los mandatos necesarios para proceder al compilado y encadenado de esta aplicación serían:

```
CLIPPER prog -m
CLIPPER modulo1
CLIPPER modulo2
```

```
CLIPPER modulo3 -m
CLIPPER submod1
CLIPPER submod2
RTLINK @encadena
```

Obsérvese que se ha empleado la opción -m en modulo3 para que no se compilen las llamadas DO. Los submódulos son compilados por separado para obtener archivos objeto distintos. La memoria quedaría del siguiente modo:

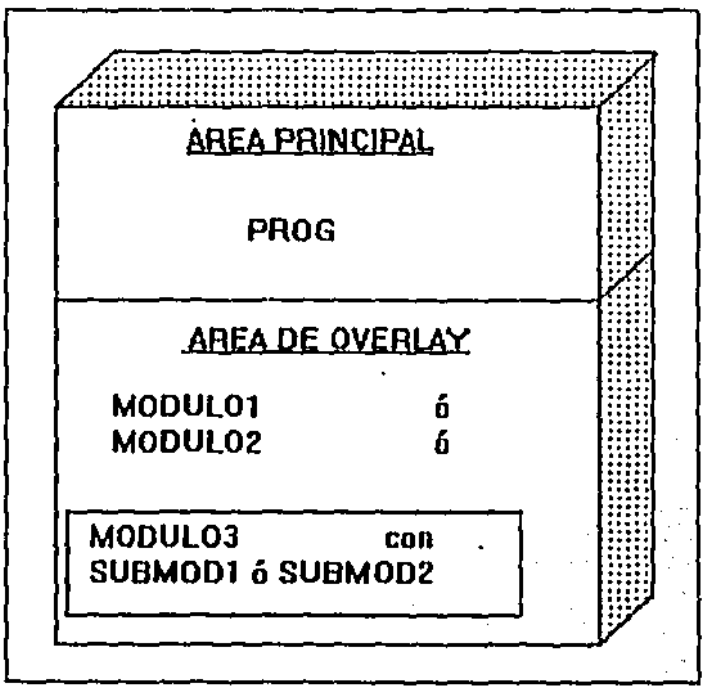


Figura 2.5

TECNICA 4

Overlays estáticos externos: Con cualquiera de las tres técnicas anteriores podemos solucionar problemas de escasez de memoria RAM. Pero también pueden usarse para resolver problemas de escasez de espacio en disco. Supongamos que nuestra aplicación debe ejecutarse en disco flexible y que el propio archivo prog.exe no

cabe físicamente en él. Hemos entonces de distribuir sus procedimientos e ir indicando al usuario cuando debe cambiar de disco flexible. Lo que puede hacerse en una situación como ésta es distribuir el contenido del archivo prog.exe en varios archivos.

Para que Clipper produzca un archivo .EXE y tantos archivo .OVL como módulos para overlay tengamos definidos, sólo hay que cambiar en cada una de las tres técnicas anteriores la instrucción:

```
SECTION FILE <ArchObj>
```

por:

```
SECTION INTO <ArchOvl> FILE <ArchObj>
```

donde <ArchOvl> es el archivo .OVL que resultará y <ArchObj> es el archivo objeto del cual lo creamos.

OVERLAYS DINAMICOS

Son aquellos en los cuales se define un área de memoria al momento de la ejecución. No existen restricciones, de tal forma que el manejador de overlays de RTLINK se ocupa de administrar la memoria a través de páginas de tamaño fijo de 1Kb.

El tamaño del área para overlays dinámicos se ajusta automáticamente según las necesidades del sistema, pero si se desea, se puede definir el tamaño mediante la variable de ambiente CLIPPER, con la siguiente sintaxis:

```
SET CLIPPER=DYNK:<NumKb>
```

Donde NumKb es la cantidad en Kb que se quiere definir para el área.

Siguiendo con la estructura de la aplicación anterior, las ordenes necesarias para crear un archivo ejecutable con sus correspondientes overlay serían

```
CLIPPER prog -m
CLIPPER modulo1
CLIPPER modulo2
CLIPPER modulo3
RTLINK DYN FI prog,modulo1,modulo2,modulo3
```

Puesto que cada uno de los módulos es compilado por separado es necesario usar la opción -m al compilar el programa principal. La opción DYN indica que el overlay será dinámico.

OVERLAY DINAMICOS EXTERNOS

Los overlay dinámicos pueden definirse como archivos externos, con extensión .OVL. Suponiendo que queremos crear un archivo .EXE con un overlay dinámico externo de nombre arch_over, cambiar la instrucción:

```
RTLINK DYN FI prog,modulo1,modulo2,modulo3
```

Por

```
RTLINK DYN INTO arch_over FI prog,modulo1,modulo2,modulo3
```

Donde la opción INTO arch_over indica que se creará un archivo .OVL de nombre arch_over.

2.2 LIBRERIAS DE CLIPPER 5.01

Clipper 5.01 proporciona las siguientes librerías:

CLIPPER.LIB	Librería estándard de Clipper
EXTEND.LIB	Una extensión de librería de Clipper
CLD.LIB	Depuración
DBFNTX.LIB	Funciones para el driver que maneja las bases de datos .DBF y los índices .NTX.
RTLUTILS.LIB	Utilidades del enlazador
TERMINAL.LIB	Funciones para el driver de terminal adecuado

Además de estas librerías suministradas por Clipper 5.01, el usuario puede crear sus propias librerías o puede adquirir librerías hechas por fabricantes independientes, por ejemplo:

DGE.LIB	Librería para gráficos
ARLIB.LIB	Funciones de uso general
LLIBCA.LIB	Librería para el manejo de variables de punto flotante.
POWER TOOLS	Funciones de uso general.
SILVER COMM	Funciones de comunicaciones para Clipper
ADCOMM	Funciones de comunicaciones para Clipper
BLINKER	Enlazador más eficiente, contempla las mismas características que RTLINK, sin embargo el proceso de encadenamiento es superior.

No es necesario que en las sesiones de enlace indique las librerías, ya que RTLINK toma por omisión todas las que necesita. Sólo indique en la cláusula LIBRARY aquellas otras librerías que son hechas por fabricantes independientes.

2.3 LIBRERIAS PREENCADENADAS

El encadenamiento es un proceso largo, sin embargo RTLINK permite reducir este tiempo generando archivos que contienen código compartido por diferentes aplicaciones, de tal manera que en los subsecuentes procesos de encadenamiento, estas rutinas no necesitan encadenarse debido a que ya existen en archivos llamados pre-linked libraries (librerías preencadenadas).

EL concepto de las librerías preencadenadas (PLL) es simple. Poner el código que no se quiere en un archivo .EXE en otro archivo con extensión .PLL. Este archivo puede contener código común para diferentes aplicaciones, con lo cual se ahorrará espacio en disco y el encadenador (RTLINK) tardará menor tiempo en ejecutar su tarea.

No se obtiene un ahorro en el espacio en disco con la elaboración de librerías preencadenadas a no ser que se tengan dos o más archivos .EXE que necesiten el mismo código del PLL. Cuando se tienen varias aplicaciones, la reducción en los archivos .EXE y por consecuencia la reducción en el espacio ocupado en disco, puede ser significante. Si se tuviera solo una aplicación, con su archivo .PLL, no habría ahorro de espacio, al contrario, ocuparían más espacio de lo normal.

Una vez que se decide crear una PLL, la pregunta obvia que surge es: ¿Que debo poner ahí?. Para responder a esta pregunta debemos decidir cuál código es el que queremos compartir, y centralizarlo en un archivo .PLL. Por razones obvias, es sumamente importante que este código sea estable.

2.3.1 Como crear librerías preencadenadas

Cuando se instala Clipper 5.01 se crea un directorio PLL. Para utilizar las librerías preencadenadas, primero debe declararse una variable de ambiente llamada PLL que contenga la ruta en donde se encuentra ese

directorio. Usar la siguiente sintaxis:
SET PLL=<RUTA> <\PLL>

Por ejemplo, pudiera ser algo como lo siguiente:

SET PLL=C:\CLIPPER5\PLL

Dentro de ese directorio hay un archivo script llamado BASE50.LNK. El contenido de este archivo es:

```
prelink
output base50

lib clipper, extend, terminal, dbfntx
refer _main
refer _VOPS, _VMACRO, VDB, _VDBF, _VDBFNTX
refer _VTERM, _VPICT, _VGETSYS
refer _VDBG
```

Para tener su conjunto de librerías preencadenadas deberá enlazar el contenido de este archivo tecleando:

RTLINK @BASE50

Esto hará que se creen dos archivos: BASE50.PLL y BASE50.PLT.

Ahora ya está listo para usar a RTLINK mediante la técnica de librerías preencadenadas. Escriba cualquier programa, compílelo y encadénelo con la sintaxis:

RTLINK FI <programa> /PLL:BASE50

Lo cual dará como resultado a un archivo .EXE que usará a BASE50.PLL.

2.3.2 Como preencadenar sus propias librerías

El sistema de las librerías preencadenadas usa las librerías estándard de Clipper. Pero es posible que en sus aplicaciones use alguna librería propia que necesite usar según esta misma técnica. Supongamos que su librería se llama libre.lib y que contiene numerosas funciones, entre ellas, una que se llama lfunc. Escriba su propio archivo .LNK conteniendo:

```
prelink
output librepre

lib clipper, extend, terminal, dbfntx, libre
refer _main
refer _VOPS, _VMACRO, VDB, _VDBF, _VDBFNTX
refer _VTERM, _VPICT, _VGETSYS
refer _VDBG
refer LFUNC
```

Donde **output** indica el nombre de salida del PLL, **lib** indica las librerías (incluimos la nuestra) y mediante **refer** mencionamos a una de las funciones de nuestra librería, Esto último es imprescindible ya que de no hacerlo, el encadenador ignora la petición de incorporar a libre.lib.

Suponiendo que el nombre del archivo .LNK es encadena.lnk, teclear:

```
RTLINK @ENCADENA
```

Con lo que se obtienen los archivos librepre.pll y librepre.plt. Para usarlos en una aplicación, sólo tiene que añadir a la sentencia de enlace la opción /PLL:LIBRE.PRE. Por ejemplo, si el nombre del programa principal fuera prog, teclear:

```
RTLINK FI prog /PLL:librepre
```

2.4 HERRAMIENTAS MAKE

Durante la creación y mantenimiento de nuestras aplicaciones el tamaño y cantidad de los archivos . .PRG puede crecer considerablemente.

Si compilamos todos los archivos fuente por separado, o en pequeños grupos de rutinas relacionadas, no tendremos necesidad de recompilar todo cada vez que hagamos un pequeño cambio en nuestra aplicación. Sólo tendríamos que checar el directorio y recompilar únicamente aquellos archivos .PRG que tuvieran fecha y hora mas reciente que sus correspondientes archivos .OBJ.

La utilería RMAKE hace esto y mas. Checa si los archivos fuente son mas viejos que sus correspondientes archivo objeto. Si el archivo fuente es mas reciente, nada sucede, pero si no es así, el archivo objeto es recompilado. Además tiene la capacidad de usar variables definidas por el usuario para controlar este proceso sin alterar el archivo MAKE (extensión .RMK).

2.4.1 La Utilería RMAKE

La utilería RMAKE tiene la siguiente sintaxis:

RMAKE [<Archivo(s)MAKE>] [<Macro(s)>] [<Switches>]

<Archivo(s)> Es uno ó mas archivos .RMK. Puede contener la unidad, la ruta y los comodines normales del DOS.

<Macro(s)> Son similares a las variables Clipper. Las macros pueden contener varias cosas, pero generalmente contienen nombres de archivo.

<Switches> Son controles similares a los switches del compilador, que determinan ciertos aspectos de RMAKE. Se puede usar el comando SET del DOS para crear una variable de ambiente llamada por RMAKE para guardar las opciones por default. Cualquier conflicto entre la variable y la invocación en la linea de comandos, es resuelta a favor de la línea de comandos.

2.4.2. La opción MAKEPATH

Para poder realizar su tarea, la utilería RMAKE necesita encontrar a todos los archivos que serán comparados. Generalmente todos los archivos se encuentran en un mismo directorio pero de no ser así, se puede usar la opción MAKEPATH. Esta opción se puede incluir dentro del archivo RMAKE y tiene una sintaxis similar a la del path del DOS:

"MAKEPATH .CH " = <ruta especificada>

Dentro del MAKEPATH se pueden usar comodines, además se puede usar una macro previamente definida. En RMAKE la macro se indica con el signo $(). Ejemplo:

"MAKEPATH.EXE" == "C:\ CLIPPER5; $ (MAKEPATH.LIB)"

2.4.3 El proceso de comparación de RMAKE

RMAKE usa un archivo ASCII con extensión .RMK para controlar el proceso de comparación. Contiene cinco diferentes elementos:

Regla de dependencia
Regla de inferencia
Macros RMAKE
Switches RMAKE
Comandos RMAKE

Regla de dependencia

Esta regla hace la comparación de fecha y hora entre los archivos elegidos y los archivos de los cuales dependen. Dependencia significa que un archivo desciende o es creado a partir de otro. Por ejemplo: los archivos objeto dependen de los archivos fuente, porque compilando los fuente se obtienen los objeto. Si el archivo fuente tiene fecha y hora posteriores al archivo objeto, entonces se realiza la compilación, en caso contrario no se hace. Las reglas de dependencia tienen el siguiente formato:

<ArchDepend> : <Arch1>[,<Arch2>,...,<ArchN>]

Los dos puntos (:) dividen los dos lados de la comparación. Si la fecha y la hora del archivo de la izquierda (ArchDepend) es mas reciente que la de los archivos de la derecha (Arch1, Arch2,...,ArchN), los comandos posteriores no son ejecutados.

Por ejemplo, si tuviéramos un archivo objeto PROG creado por la compilación de los archivos PROG1, PROG2 y PROG3, el archivo .RMK sería algo como lo siguiente:

```
# Archivo RMAKE para PROG.EXE
PROG.OBJ : PROG1,PROG2,PROG3
CLIPPER PROG1,PROG2,PROG3
```

En este ejemplo la primera línea solamente es un comentario. En la segunda línea, RMAKE examina si alguno de los tres archivos de la derecha es mas reciente que el de la izquierda. Sólo si es así, la tercera línea es ejecutada y sólo el(los) archivo(s) con fecha y hora mas reciente es(son) recompilado(s).

Regla de inferencia

La regla de inferencia determina la acción a ser tomada después de que los nombres de archivo son substituidos dentro de la fórmula. La

regla de inferencia se crea con la siguiente sintaxis:

<ArchDest>.<ArchIni>:
<Comando>

Donde ArchDest es el archivo que se crea a partir de ArchIni. Por ejemplo:

.PRG.OBJ:
CLIPPER $* -M

La macro $* substituye el nombre del archivo RMAKE menos la extensión. Las macros permiten manejar reglas de dependencia y de inferencia en forma variable, asociando una cadena de caracteres a un nombre especifico. Esta regla de inferencia podría hacer uso de una tabla como la siguiente:

Prog1.obj : Prog1.prg
Prog2.obj : Prog2.prg
Prog3.obj : Prog3.prg
Prog4.obj : Prog4.prg

La utilería RMAKE substituye el programa objeto y archivos especificados en la tabla dentro de la regla de inferencia, uno a la vez. Si el archivo fuente es mas reciente que el objeto, se realiza la compilación. La macro $* substituye el nombre del archivo en la tabla.

Macros RMAKE

RMAKE usa una lista especial de símbolos para macros:

$^ Retorna un nombre de archivo menos su extensión
$@ Retorna un nombre de archivo con extensión
$** Retorna una lista de nombres de archivos dependientes

Por ejemplo, se pueden crear macros que esperen por grupos de nombres de archivos, tecleando lo siguiente en el archivo RMK:

var = archivo1.obj archivo2.obj archivo3.obj archivoN.obj

El archivo .RMK puede substituir $(var) a los nombres de archivos. A continuación se da un ejemplo del contenido de un archivo .RMK que muestra como las macros son usadas para identificar archivos para la utilería RMAKE.

```
// Crea dos macros para los archivos dependientes
part1 = menu.obj banner.obj
part2 = net_udfs.obj cliente.obj

// Crea la regla de inferencia para la compilación
.prg.obj
Clipper $* /m
// la macro $* substituirá uno de los nombres de archivos en la
// siguiente sección de selección

// Lista de archivos a ser comparados
menu.obj        :  menu.prg
net_udfs.obj    :  net_udfs.prg
cliente.obj     :  cliente.prg

// Si menu.obj es mas antiguo que menu.prg, el comando
// Clipper /m  será puesto en el archivo de lotes.

// Esta es una regla de dependencia para manejar una subrutina
// escrita en ensamblador
banner.obj  :  banner.asm
MASM banner

// Esta es una regla de dependencia para controlar el proceso de
// encadenado:
menu.exe : $(part1) $(part2)
RTLINK FI $** LIB Clipper, extend
```

Switches RMAKE

Los siguientes switches pueden ser usados para alterar el camino del proceso RMAKE. Deben de ser precedidos por una diagonal (/) y pueden ser empleados desde la línea de comandos del DOS ó ser incluídos en el archivo RMAKE.

/B Despliega información sobre el depurado (Debugging)

/D Define una macro. Por ejemplo: /D mac1 ="prog.prg"

/I Ignora errores. Con esto se evita que termine el RMAKE antes de tiempo.

/N Hace un nulo. No realiza acciones, pero despliega acciones que deben ser tomadas.

/S Extiende la búsqueda de archivos en cualquier subdirectorio del directorio actual, así como en los especificados en las macros del MAKEPATH.

/U Permite usar el símbolo # para mostrar comentarios. Cuando esta opción es usada, todas las directivas deben usar la sintaxis del !ifdef en lugar de la normal #ifdef.

/W Despliega mensajes de bajo nivel.

Comandos RMAKE

RMAKE tiene un pequeño conjunto de comandos que permite establecer ciertas condiciones. Propiamente planeado, un archivo puede ser usado para obtener diferentes tipos de salida. Los comandos de RMAKE son parecidos a los del preprocesador. La tabla 2.2 tiene una lista de comandos RMAKE.

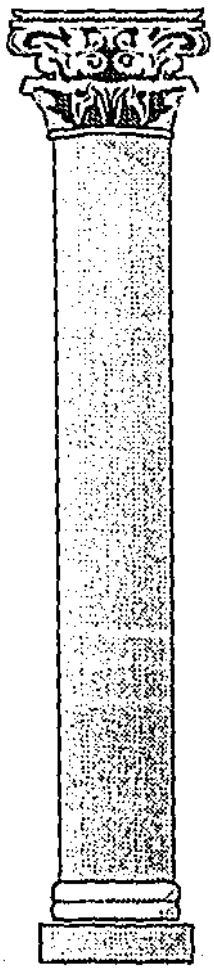
SINTAXIS	DESCRIPCION
!ifdef <nomb_macro> <sentencias> [!else <sentencias>] !end	Este if es similar al del lenguaje Clipper. Si el nombre de la macro "nomb_macro" está definido, la condicional es verdadera. En caso contrario se ejecutan las sentencias correspondientes al !else (si es que existe).
!ifndef <nomb_macro> <sentencias> [!else <sentencias>] !end	El !ifndef es exactamente lo opuesto al !ifdef. Si el nombre de la macro no está definido, la condicional es verdadera.
!ifeq <token1><token2> <sentencias> [!else <sentencias>] !end	Con este if se toma una decisión dependiendo si los tokens son iguales o no. Los tokens comparados pueden ser cadenas de caracteres o contenidos de macros. Para que este token se considere verdadero ambos tokens deben ser exactamente iguales.
!iffile <nomb_arch> <sentencias> [!else <sentencias>] !end	Esta estructura permite saber si un archivo o archivos existen. La especificación de archivo(s) permite comodines.
!Undef <nomb_macro>	Este comando elimina una macro de la lista de macros. No existe una forma para borrar una variable de ambiente DOS, pero se puede definir una macro con ese nombre para ocultarla.
!include <nomb_arc>	Este comando permite especificar otro archivo para ser substituído dentro del archivo .RMK en una locación en particular. Las entradas del archivo especificado serán procesadas como si fueran parte del archivo .RMK.

Tabla 2.2 Lista de comandos de Rmake. (Continuá).

!msg <token>	Este comando despliega un mensaje respecto al progreso del proceso. El token debe ir encerrado entre comillas (").
!! <acción>	Es similar al comando Run del lenguaje Clipper. Ejecuta cualquier comando del DOS. Al terminar esta ejecución, el make continúa

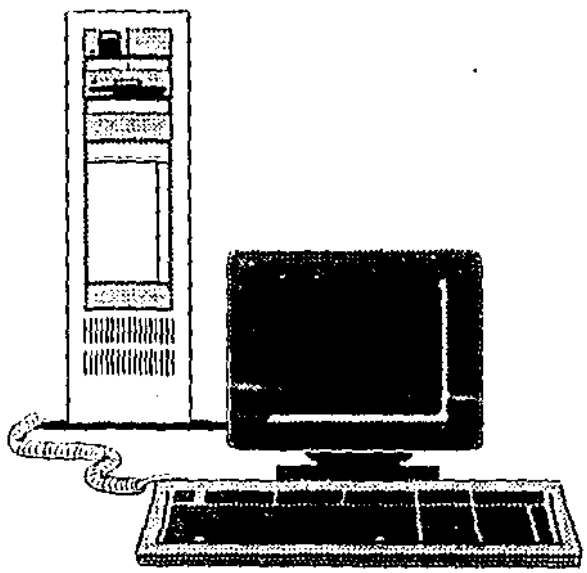
Tabla 2.2 Lista de comandos de Rmake. (Continuación).

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO



CAPITULO 3

- DECLARACION DE VARIABLES Y TIPOS DE DATOS



ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES
ARAGON

3.1 ATRIBUTOS DE LAS VARIABLES

Para programar en Clipper o en otro lenguaje, es necesario entender bien el funcionamiento y la utilidad de las variables de memoria.

¿Qué son las variables de memoria?

Una variable de memoria es un una porción de memoria de la computadora que recibe un nombre y almacena temporalmente algún dato o información.

Una variable no existe hasta que alguna porción de la memoria de la computadora contenga su valor, algunas variables son creadas automáticamente mientras otras deben ser explícitamente declaradas, dependiendo del tipo de variable, unas siempre existirán en memoria, otras se borrarán automáticamente y algunas deberán ser explícitamente eliminadas. La duración en memoria está relacionada con el tiempo de vida de una variable.

Una variable de memoria puede contener números, cadenas de caracteres, una fecha o un valor lógico .T. (verdadero) o .F. (falso). A partir del momento en que se crea una variable, no necesita hacer referencia a su contenido, pero sí a su nombre. Ver la figura 3.1.

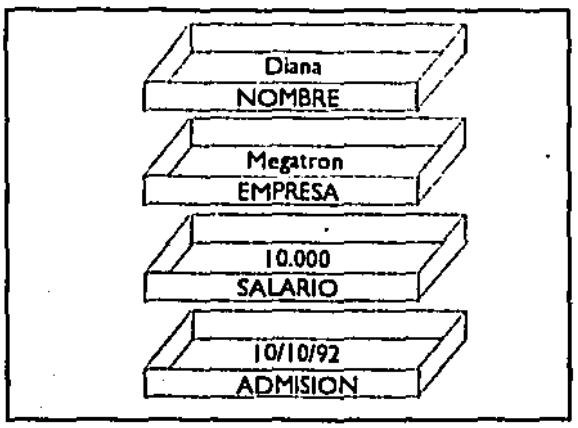


Figura 3.1 Variables de Memoria

El nombre de una variable puede contener hasta diez caracteres. Sigue las mismas reglas de creación de nombres de campos. Se permite la creación ilimitada de variables, dependiendo únicamente de la memoria disponible.

El nuevo concepto del manejo de variables en CLIPPER, proporciona un elemento sumamente valioso para el desarrollo la *"visibilidad"*.

Antes de entrar a este punto, definamos a una variable en función a tres atributos, que son:

- Tipo
- Visibilidad
- Vida

3.1.1 Tipos De Datos

En la versión sumer `87 se manejaban diferentes tipos de datos; numérico, lógico, carácter, fecha, y memo, en la versión de CLIPPER 5.01 se añadieron tres tipos de datos: bloques de código, (code blocks), Nil, y Objetos.

Todas las variables, excepto las públicas se inicializan con el valor "Nil".

Si utilizamos la función VALTYPE() (que es una mejora de la función TYPE()), para determinar los diferentes tipos de datos que soporta Clipper 5.01. La tabla 3.1 muestra los valores de regreso.

Carácter Puede contener letras, números y símbolos de puntuación del teclado más algunos gráficos (dibujo de cajas), alfabetos extranjeros y caracteres especiales de símbolos (p.e. þ, ø, _, etc.). Un campo del archivo de datos y una variable de memoria de Clipper puede contener hasta 32 Kb de caracteres.

CODIGO	REGRES	SIGNIFICAD
VALTYPE(Arreglo)	A	Arreglo
VALTYPE(Block)	B	Code Block
VALTYPE(Carácter)	C	Carácter
VALTYPE(Fecha)	D	Fecha
VALTYPE(Lógico)	L	Lógico
VALTYPE(Memo)	M	Memo
VALTYPE(Numérico)	N	Numérico
VALTYPE(Objeto)	O	Objeto
VALTYPE(Nulo)	U	Nil

Tabla 3.1

Fecha — Puede contener, números que representan una fecha. En la base de datos los campos tipo fecha siempre tienen una anchura de ocho caracteres, pero su formato de impresión puede variar. Por ejemplo, se puede especificar que las fechas aparezcan en formato americano (mm/dd/aa), italiano (dd-mm-aa) o ANSI (aa.mm.dd). Las fechas también se pueden visualizar con los cuatro dígitos en el año (mm/dd/aaaa).

Lógico — Puede contener las letras T (verdadero) o F (falso) para indicar que una condición es verdadera o falsa. También se puede introducir Y o N, pero se almacenan como T o F.

Memo — Puede contener cualquier letra, número y símbolos de puntuación del teclado más algunos gráficos (dibujo de cajas), alfabetos extranjeros y caracteres especiales de símbolos (p.e. þ, ø, _, etc.). La longitud máxima de un campo memo es de 64 Kb.

Las variables obtienen su tipo al momento en que se les asigna un valor, por ejemplo:

```
lo_nCon:= 1
lo_cCve:= [AÑO2000]
```

en donde "lo_nCon" es una variable de tipo numérico y "lo_cCve" es una variable de tipo carácter.

En Summer 87, no existían los tipos de datos "Block", "Objeto" y "Nulo", el análisis de éstos se efectuarán en tópicos posteriores al igual que los arreglos.

3.1.2 Visibilidad De Variables

La visibilidad de una variable tiene que ver de acuerdo a la declaración de la misma. El concepto de visibilidad se aplica a la forma en la que una variable puede ser accesada, es decir, algunos módulos o rutinas tienen acceso a las locales a su definición, sin embargo no son capaces de ver o accesar las de otros módulos (a menos que sean "públicas").

Clipper soporta cuatro diferentes tipos de declaración de variables y todas las variables, excepto las públicas se inicializan con el valor "Nil":

Private Estas variables son creadas al momento de la ejecución. Existen tres formas de inicializar una variable como privada, con el comando PRIVATE, con el comando PARAMETERS o con la asignación normal de la variable, el comando DECLARE utilizado para arreglos, indica que la variable de arreglo será de tipo privada. Una vez declarada la variable, está permanecerá en memoria durante la ejecución de la función o procedimiento que la inicializó y cualquier otra función o procedimiento a niveles inferiores tendrá acceso a esta variable. Por otra parte las funciones y procedimientos en niveles superiores no tendrán acceso, de hecho no existirá en memoria. Si una variable de tipo privada es declarada al inicio de una aplicación su comportamiento será idéntico a la variable de tipo PUBLIC.

Public Estas variables son creadas al momento de la ejecución. Una vez inicializada con el comando PUBLIC, a la variable se le asignará automáticamente con un valor lógico igual a .F., valor que permanecerá hasta que a la variable se le asigne un tipo diferente. Estas permanecen en memoria durante la ejecución

de la aplicación, y podrán ser modificadas, consultadas o simplemente manipuladas por cualquier procedimiento o función en cualquier nivel de la aplicación, puede ser eliminada de memoria por cualquiera de los siguientes comandos: CLEAR ALL, CLEAR MEMORY, ó RELEASE <variable>.

Local Estas variables son inicializadas con un tipo nulo igual a Nil, y son creadas al momento de la compilación. La visibilidad de una variable local depende del lugar en donde se haya definido, es decir, puede se local a todo un archivo que contiene una serie de rutinas, o bien puede se local a una rutina. En su código fuente se deberá de especificar el comando LOCAL, como se muestran en el ejemplo siguiente:

```
FUNCTION Hora()
LOCAL lo_var:= 10
.
.
código1
Calcula()
.
.
código2
RETURN(.T.)
```

La variable lo_var fue declarada como local, definida con un valor de 10, cuando la función Calcula() es ejecutada lo_var existe, pero no puede ser accesada, y continuará existiendo hasta que termine la ejecución de la función Hora(), lo_var será visible tanto para el código 1 como para el código 2, pero no para el código de Calcula(); si se accesa nuevamente la función Hora(), la variable lo_var será definida nuevamente.

Static Estas variables, al igual que las locales, son inicializadas con un tipo nulo igual a "Nil", y son creadas al momento de la compilación. La visibilidad de una variable estática depende del lugar en donde se haya definido, es decir, puede ser local a todo

un archivo que contiene una serie de rutinas, o bien pude ser local a una rutina. Las variables estáticas difieren de las locales en el sentido de conservación del valor, debido a que una vez que se ha inicializado una variable estática, ésta conserva su valor aún y cuando termine la ejecución de la función que la definió. En su código fuente se deberá especificar con el comando STATIC, ejemplo:

```
FUNCTION Hora()
STATIC lo_var:= 10
.
.
código1
Calcula()
.
.
código2
RETURN(.T.)
```

La primera vez que se ejecuta la función Hora(), lo_var se inicializa con un valor de 10, el cual podrá ser visualizado por código 1 y código 2 y no por el código de Calcula(), pero entonces ¿Cuál es la diferencia? Supongamos que en el código 2 se encuentra la línea:

```
lo_var:= lo_var +1   que es idéntica a  lo_var++
```

la siguiente vez que llame a Hora() el valor de lo_var será 11, es decir no se definirá nuevamente, como en el caso de las variables locales, y cada vez que llame a la función Hora(), el valor de lo_var será incrementada en uno.

En resumen las variables locales siempre se inicializan cada vez que llame a la función o procedimiento, y la variable conservará su valor solamente durante su ejecución; las variables estáticas se inicializan solamente la primera vez que la función o procedimiento sea llamado y no pueden ser borradas de memoria, su vida se inicia en la declaración y termina al finalizar la ejecución del sistema.

3.1.3 Vida De Las Variables

La vida de las variables se refiere al concepto de tiempo desde su creación, hasta que desaparece. Este concepto difiere de la visibilidad, debido a que una variable puede terminar su visibilidad o acceso y sin embargo mantener su valor y por lo tanto su existencia.

Este concepto muestra claramente lá diferencia entre variables locales y las estáticas. Ambas son iguales a excepción de que las variables estáticas una vez que han sido definidas, mantienen su valor hasta que sea destruida explícitamente, observe el código siguiente:

```
Function Contador
  Static lo_nCont:= 1
  lo_nCont++
Return(lo_nCont)
```

la primera vez que se ejecute esta función, se creará la variable "lo_nCont" con un valor inicial de 1. La siguiente vez que se ejecute esta función, "lo_nCont" ya existe con el último valor asignado de tal forma que no se vuelva a crear.

Dentro de la declaración de variables se puede utilizar la asignación como se muestra en el ejemplo anterior.

Por otra parte, las variables locales y estáticas son más rápidas de accesar que las públicas y las privadas, y utilizan menos memoria. La siguiente tabla muestra la visibilidad y vida de las variables en función a la declaración:

DECLARACION	VISIBILIDA	VIDA
Local	Local	Rutina de creación
Private	Privada	Rutina de creación
Public	Global	Global
(External)Static	Módulo	Global
(Internal)Static	Local	Global
---------	Privada	Rutina de creación

TABLA 3.2

3.2 INICIALIZACION DE VARIABLES

Las variables pueden ser inicializadas de dos formas:

- En la declaración
- Como estatuto

Cuando se declara una variable (asignación de visibilidad), aún no se tiene definido el tipo, por ejemplo:

```
Local lo_nTotal, lo_nIva
Local lo_cFile
```

en los estatutos, se declarán las variables lo_Total, lo_nIva, lo_cFile, aún y cuando se ha definido la visibilidad de dichas variables, estas no tienen un tipo específico.

Esto significa que la definición del tipo de variables se da en el momento en el que se inicializan. Por ejemplo:

```
Local lo_nTotal, lo_nIva
Local lo_cFile
lo_nIva:= 10
```

en este caso, la única variable que tiene un tipo definido es "lo_nIva" como numérica.
Por otra parte, es posible declarar una variable asignándole sus atributos en un solo estatuto, por ejemplo:

```
Local lo_nTotal:= 0, lo_nIva:=10, lo_cFile:= "Equipos"
```

en donde las variables "lo_nTotal" y "lo_nIva" son de tipo numérico con una visibilidad local (al archivo o al procedimiento), y la variable "lo_cFile" de tipo carácter con una visibilidad local.

Estas reglas de inicialización se aplica a todas las variables de los tipos que soporta CLIPPER.

3.3 VALORES NIL

CLIPPER soporta un tipo de datos llamados "NIL". Este tipo de datos no tiene representación. Cuando declaramos variables sin inicializar, estas toman por default el valor Nil.

El valor Nil de una variable significa que no tiene dirección asignada en memoria, es decir, que no existe un apuntador, ejemplo; cuando declaramos una variable y la definimos de tipo carácter, entonces esta variable es una dirección de memoria de donde podemos accesar la información.

Suponiendo que tenemos el siguiente código:

```
Local lo_cNom:= "I.M.P."
Local lo_nCon
```

Viendo un mapa de memoria tendríamos lo siguiente figura 3.2.

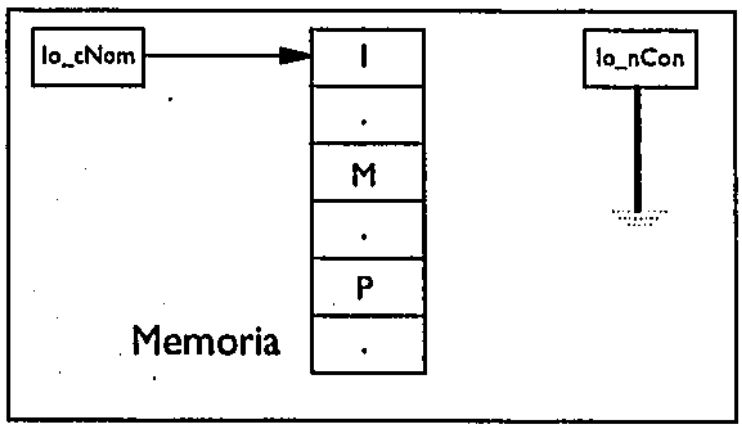


Figura 3.2.

3.4 OPERADORES Y EXPRESIONES

CLIPPER 5.01 cuenta con nuevos operadores, que puede decir que son heredados del lenguaje C, y uno de Pascal.

Desde luego que puede manejar los mismo operadores que la versión Summer ´87, sin embargo con la inclusión de estos operadores, la construcción de expresiones se convierte en un elemento poderoso y elegante para el desarrollo de aplicaciones.

La tabla 3.3. muestra los operadores que soporta CLIPPER 5.01.

Nuevos operadores:

Clipper 5.01 incorpora nuevos operadores; todos los operadores que utilizaba la versión de Summer '87, pueden ser utilizados en al nueva versión.

Se introdujo un concepto muy conocido en el lenguaje C, que son los operadores compuestos, estos operadores realizan una operación entre dos operandos. Con la inclusión de estos operadores, la construcción de expresiones se convierte en un elemento poderoso y elegante para el desarrollo de aplicaciones. Tabla 3.4

DESCRIPCION	OPERADOR	NOMBRE
Operadores Heredados de Summer 87	u	Operador unitario
	** ó ^	Exponenciación
	*	Multiplicación
	/	División
	%	Módulo
	+	Suma
	-	Diferencia
	=	Asignación
	++	Incrementador unitario
Operadores Heredados del Lenguaje C	--	Decrementador unitartio
	+=	Suma compuesta
	-=	Diferencia compuesta
	*=	Multiplicación compuesta
	/=	División compuesta
	%=	Módulo compuesto
	**	Exponenciación compuesta
Operador Heredado de Pascal	:=	Asignación (dos puntos igual)
Operadores Relacionales	<	Menor que
	>	Mayor que
	<=	Menor o igual que
	>=	Mayor o igual que
	==	Igual a
	!= ó <>	Diferente a
Operadores Lógicos	.OR.	Unión
	.AND.	Disyunción
	.NOT. ó !	Negación

TABLA 3.3 Operadores

Al igual que en el operador de incremento, el operador de decremento utiliza las mismas reglas, es decir, dependiendo de la posición del operador, éste se decrementará antes o después de la evaluación de la totalidad de la expresión.

Operador	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
^=	a^ = b	a:= a ^ b

TABLA 3.4. Operadores compuestos

Operadores de incremento y decremento:

Técnicamente estos operadores son de asignación:

"++" operador de incremento unitario
"--" operador de decremento unitario

```
lo_var := lo_var + 1
++lo_var
lo_var++
```

Las tres expresiones tienen el mismo resultado, incrementan en uno el valor de lo_var. Ahora bien, la posición del operador es importante en el sentido de que si el operador se encuentra a la derecha de la variable, el efecto de incrementar a la variable antes de que se evalúe la totalidad de la expresión, con ello tenemos dos casos, el incremento anterior a la totalidad de la evaluación y el incremento posterior.

```
lo_i := 10
lo_suma:= ++lo_i          (resultado 11)
```

Para este caso, primero se evalúa el valor de lo_i en uno, después se evalúa la totalidad de la expresión.

```
lo_i:= 10
lo_suma:= lo_i++          (resultado 10)
```

El valor lo_i es incrementado hasta que se evalúe la totalidad de la expresión, al ser evaluada la expresión lo_i tendrá el valor de 11.

Así como el sumar la unidad a un número, también podemos restar la unidad.

```
lo_var:= lo_var -1
--lo_var
lo_var--
```

Asignación y Comparación:

Si bien es cierto que podemos utilizar el operador "=" como asignación, Clipper 5.01 incorpora un nuevo operador ":=". La diferencia de utilizar este nuevo operador como asignación se puede ver en la siguiente expresión:

```
Primer caso
WHILE (lo_i = Calcula() ) <> Nuevo()
.
.
.
END

Segundo caso
WHILE (lo_i := Calcula() ) <> Nuevo()
.
.
.
END
```

En el primer caso, la variable será comparada con en resultado de la función Calcula() debido a la connotación del estatuto WHILE; y posteriormente el resultado de dicha comparación se compara con el resultado de la función Nuevo().

En el segundo caso, la variable lo_i se le asigna el resultado de la función Calcula(), y posteriormente se compara con el resultado de la función Nuevo().

Como recomendación, utilice el operador ":=" para asignación y "==" para comparación.

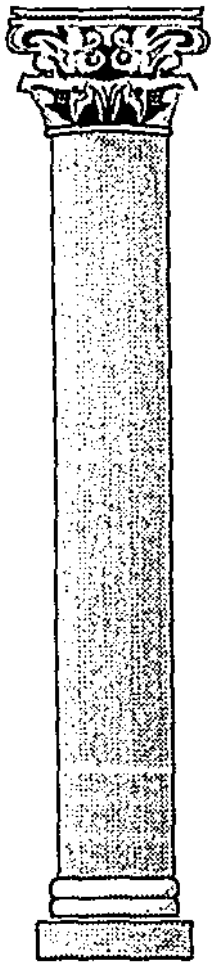
Comentarios:

Anteriormente se utilizaba "&&" y "*" para hacer un comentario en una línea. Se introducen dos formas nuevas:

/* comentario */
// comentario

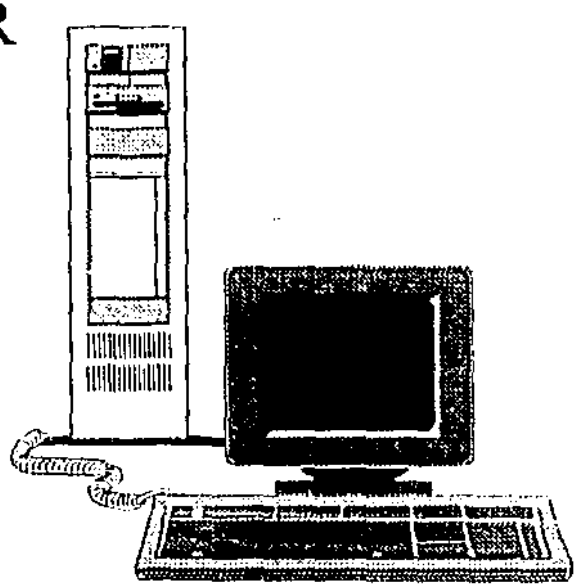
En la primera forma el tamaño del comentario entre "/*" y "*/", puede ser tan largo como se desee y ocupar cualquier número de líneas. La segunda forma es utilizada en la misma manera que "&&" o "*".

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO



CAPITULO 4

- EL PREPROCESADOR DE CLIPPER



ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES
A R A G O N

4.1 QUE ES EL PREPROCESADOR DE CLIPPER 5.01

El preprocesador es un nuevo e importantísimo concepto en Clipper. Cuando usted invoca a Clipper.exe para compilar sus programas, dos pasos son desempeñados. Primero el código es preprocesado y después es compilado. En el primer paso busca determinadas directivas y las traduce en código estándar de Clipper. Este código es introducido en un archivo con el mismo nombre del fuente y la extensión .PPO. Una vez realizada la fase de preproceso se produce la fase de compilación. El archivo .PPO no se destruye al terminar la compilación si usamos la opción /P del compilador. Veamos el siguiente programa de ejemplo:

```
#define ESCAPE 27
tecla :=0
WHILE (tecla <>ESCAPE)
        tecla=INKEY (0)
END
RETURN
```

En el usamos una directiva para el preprocesador:

```
#define ESCAPE 27
```

Esto indica que cada vez que en nuestro programa anotemos ESCAPE lo que realmente queremos indicar es 27. Compilemos ahora nuestro programa con la opción /P. Esto hará que al terminar la compilación tengamos sobre nuestro directorio un archivo con el mismo nombre que hayamos dado al archivo fuente y con la extensión .PPO. Edítelo y verá que su contenido es:

```
tecla :=0
WHILE (tecla <>27)
    tecla=INKEY (0)
END
RETURN
```

Como puede observar, lo que ha hecho Clipper 5.01 en la fase de preproceso ha sido traducir las directivas del preprocesador. Crea un archivo fuente nuevo que es el que realmente se compila.

4.2 DIRECTIVAS DEL PREPROCESADOR

Las directivas del preprocesador son las órdenes que hemos de anotar en nuestros programas fuente para indicar a Clipper 5.01 que es lo que debe hacer en la fase de preproceso previa a la compilación. Las directivas del compilador de Clipper 5.01 son:

#command, #translate
#define
#ifdef
#include
#undef
#error

4.2.1 Directivas #command y #translate

La utilidad de las directivas #command y #translate consiste en la posibilidad que ofrecen de reducir a una frase, de sintaxis comprensible, todo el código de una oscura función de usuario. La sintaxis de ambas es:

#command <patrón a definir> =><patrón resultante>
#translate <patrón a definir> =><patrón resultante>

El mandato definido es <patrón a definir>. Cada vez que en nuestros fuentes empleamos <patrón a definir>, este se substituye por <patrón resultante> que debe ser ya una función presente en alguna de las librerías de Clipper. No confundir el "=>" con el signo mayor ó igual que.

MARCA	USO
<marca>	Es el tipo estándard de marcador de concordancia entre un mandato y su definición.
<marca,...>	Es una lista de marcadores estándard de concordancia.
<marca:lista>	Marcador de concordancia restringido. Si no se anota una claúsula de lista, se pregunta por la siguiente.
<*comodin*>	Se usa para representar el texto que se anota desde la última posición de un mandato.

Tabla 4.1 Marcas de concordancia para el "patrón a definir"

MARCA	USO
<marca>	Es el tipo estándard de marcador de concordancia entre un mandato y su definición.
#<marca>	Sirve para tomar texto desde una marca de <patrón a definir>. Si no existe ningún texto se asume la cadena nula.
<"marca">	Sirve para tomar texto desde una marca de <patrón a definir>. Si el texto no se escribió entre paréntesis no se asume nada.
<(marca)>	Sirve para tomar texto desde una marca de <patrón a definir>. Si el texto no se escribió entre paréntesis no se asume nada.
<{marca}>	Sirve para tomar las marcas del <patrón a definir> y transformarlas en un Code Block sin argumentos.
<.marca.>	Sirve para tomar el valor .T. cuando se ha escrito algo en el correspondiente <patrón a definir>. En caso contrario se escribe .F. y en ningún caso se asume el propio texto.

Tabla 4.2 Marcas de concordancia para el "patrón resultante"

Las directivas #command y #translate son similares en su uso. Se diferencian en que el patrón de entrada de la primera se usa sólo para instrucciones completas y la segunda puede emplearse para partes de una instrucción como claúsulas de la misma.

Las definiciones hechas con las directivas de Clipper 5.01 sólo son validas para el archivo .PRG que las contiene. Si queremos que su amplitud sea mayor debemos incluirlas en el archivo de cabecera STD.CH1 o en el alternativo al mismo que empleamos con la opción /U del compilador (ver la directiva #include).

<patrón a definir> y <patrón resultante> pueden contener varios elementos aparte de texto. Así, por ejemplo, pensemos que definimos un mandato que hace una pausa con opción a poner un mensaje cualquiera. Podemos escribir el siguiente programa:

```
#command espera <mensa> => wait <mensa>
CLEAR
QUOT("Fin del proceso ")
espera "...Cualquier tecla para continuar"
RETURN
```

Definimos el mandato espera junto con un parámetro. El parámetro es <mensa> y lo usamos para hacer que concuerden en el mandato y en la función escrita para el preprocesador. Estas marcas de concordancia se pueden ver en las tablas 4.1 y 4.2 para <patrón a definir> y para <patrón resultante> respectivamente.

Es importante aclarar que cualquier parte del <patrón a definir> que se encierre entre corchetes "[]" será siempre opcional. Pero si los corchetes aparecen en el <patrón resultante>, esto implicará que la marca que esté entre ellos se repetirá tantas veces como aparezca su marca concordante en el < patrón a definir >.

Veamos a continuación algunos ejemplos tomados directamente del archivo de cabecera estándar de Clipper 5.01, STD.CH, en los cuales se definen comandos que no existen en Clipper 5.01 para fines de compatibilidad con dBase. En la etapa de preproceso cada uno de ellos será reemplazado por un renglón en blanco.

```
#command SET ECHO      <*x*> =>
#command SET HEADING   <*x*> =>
#command SET MENU      <*x*> =>
#command SET STATUS    <*x*> =>
```

Clipper 5.01 usa la técnica del preprocesador para definir todos sus mandatos. Realmente todos los elementos de Clipper 5.01 son funciones presentes en sus librerías. Los mandatos se definen en un archivo STD.CH que el compilador incluye, por defecto en todos nuestros programas para que la sintaxis de los mandatos que escribamos sea entendida.

Si queremos crear nuestros propios mandatos no es recomendable allterar a STD.CH. Lo apropiado sería crear un archivo de cabecera .CH propio. Al compilar se usaría la opcion /U para indicar que en lugar del archivo estándar donde se definen los mandatos de Clipper, se emplee el nuestro.

4.2.2 Directiva #define

Sirve para definir una constante o una pseudofunción. La sintaxis para ello es:

#define <idconstante> [<salida>] ó

#define <función> (<argu>) [<exp>]

Donde:

<idconstante> es el nombre de uno de los identificadores a ser definido.

<salida> es el texto que substituíra a <idconstante>.

<función> es una pseudofunción con un argumento opcional <argu>. <exp>es la expresión que substituíra a la pseudofunción.

La sentencia #define es quizá la más usada en el proceso de codigo. Su utilidad viene de dos aspectos distintos:

a) Resuelve en tiempo de compilación asignaciones que en otro caso deberían resolverse en tiempo de ejecución. Es la diferencia entre:

```
#define ESC 27          y

ESC=27
```

En el segundo caso asignamos el valor constante 27 a una variable denominada ESC. Esto, por supuesto, se resuelve en en tiempo de ejecución e incrementa el número de variables que estamos usando.

En cambio, la instrucción del primer caso se resuelve en tiempo de compilación y no supone ninguna carga para el funcionamiento de la memoria.

b) Sirve para hacer más claro el código que escribimos, ya que nos permite sustituir instrucciones como:

```
IF  LASTKEY( ) =27
                          por
IF  LASTKEY( ) =ESC
```

El modo en que el procesador usa las directivas consiste simplemente en buscar a lo largo de todo nuestro programa cada una de las <idconstante> definidas y reemplazarlas por el valor de <salida> de la definición.

Las constantes definidas con #define son sensibles a la diferencia entre mayúsculas y minúsculas, siendo ésta la única situación a lo largo de Clipper en que esto sucede.
Veamos algunos ejemplos:

a) Definición de constante manifiesta:

#define F1 28

b) Definición de pseudo-función:

#define suma(x,y) x+y

En general pueden expresarse como pseudo - funciones todas aquellas cuya definición pueda albergarse en una línea de código.

c) Definición de identificador para compilación condicional.

Supongamos que tenemos un programa en el cual para realizar la compilación nos basamos en la existencia de un identificador de nombre CONDIC. Para declararlo teclearíamos:

#define CONDIC

Con la definición del identificador <CONDIC> podemos luego controlar que determinados bloques de código sólo se compilen si dicho identificador está definido.

4.2.3 Directiva #ifdef

Sirve para indicar al compilador que el código comprendido entre #ifdef y #else o #endif sólo se procese si existe el correspondiente <identificador>. Su sintaxis es:

```
#ifdef <identificador>
        <instrucciones> . . .
[#else
        <instrucciones> . . .]
#endif
```

Siguiendo con el ejemplo del identificador CONDIC:

```
#ifdef  CONDIC
        <instrucciones>. . .
#endif
```

El identificador CONDIC pudimos haberlo definido con #define, o mediante la opción /D del compilador:

Clipper <programa> /D CONDIC

4.2.4 Directiva #ifndef

Hace lo mismo que #ifdef, pero sólo si el identificador no está definido. Tiene la misma sintaxis.

4.2.5 Directiva #include

Incluye un archivo fuente en el programa donde se usa. Lo normal es que se trate de un archivo de cabecera (header file) con extensión .CH que sólo contenga directivas para el proceso, no obstante, también se puede incluir cualquier archivo fuente .PRG. La sintaxis adecuada es:

#include "<Archivo.ch ó .prg>"

El archivo a incluir puede tener otras órdenes #include, todo ello hasta 16 niveles de profundidad.

Es importante que las directivas del compilador se incluyan en archivos de cabecera en lugar de hacerlo en el propio fuente, dejando para éste la sentencia #include que llame a dicho archivo de cabecera. Con el fin de no dispersar mandatos o constantes a lo largo de nuestro código, teniéndola centrada en un único archivo donde resulta más simple mantenerla.

Clipper 5.01 aporta los siguientes archivos de cabecera (tabla 4.3) que podemos incluir en nuestras aplicaciones (se encuentran en el directorio include).

NOMBRE	TAMAÑO (Bytes)	RELACIONADO CON
ACHOICE .CH	733	Función ACHOICE
BOX .CH	764	Mandato @BOX
DBEDIT .CH	607	Función DBEDIT
DBSTRUCT .CH	286	Función DBSTRUCT
DIRECTRY .CH	301	Función DIRECTRY
ERROR .CH	1071	Errorsys
FILEIO .CH	1290	Funciones de bajo nivel
GETEXIT .CH	301	Estado del objeto GET
INKEY .CH	6434	Valor de las teclas devuelto por INKEY
MEMOEDIT .CH	920	Función MEMOEDIT
RESERVED .CH	17441	Verificación de definición de funciones.
SET .CH	1256	Función SET
SETCURS .CH	413	Función SETCUR()
STD .CH	46450	Definición del lenguaje Clipper
FRMDEF .CH	2296	Generación de informes REPORT FORM
LBLDEF .CH	1041	Generación de etiquetas LABEL FORM

Tabla 4.3 Los archivos de cabecera de Clipper 5.01

Los archivos de la tabla anterior contienen directivas para el procesador. Si se desea pueden ser editados para cambiar el nombre de los mandatos o de las funciones definidas.

El archivo STD.CH es algo especial con respecto al resto. En este archivo está contenida toda la definición de mandatos en Clipper empleando las directivas #command, #translate y #define. Este archivo se emplea por default por el compilador sin necesidad de que le indiquemos que lo haga.

4.2.6 Directiva #undef

Retira una definición realizada con #define. La sintaxis adecuada para su uso es:

#undef <identificador>

Así, si hemos realizado la asignación:

#define F1 28

Podemos anularla en cualquier momento usando:

#undef F1

El propósito de ésta directiva es evitar que el compilador envié advertencias "warnings" cuando se necesite redefinir un identificador.

4.2.7 Directiva #error

Esta directiva fue añadida con la actualización de Clipper 5.0 a 5.01. De hecho, no viene documentada en el manual de referencia.

Si #error es encontrado por el compilador, causará como efecto que el proceso de compilación se interrumpa, enviando el error C2074.

4.3 TECNICAS DE MANEJO

Las técnicas de manejo del preprocesador están relacionadas directamente con el estilo de programación que se utilice. Si usted explota adecuadamente los preprocesadores, obtendrá:

- Código no duplicado
- Manejo de constantes y valores por default (Ahorro de memoria)
- Ejecución dinámica de la aplicación (Demos).

Para aprovechar las ventajas del preprocesador de Clipper 5.01, hay que considerar las siguientes características:

- Constantes Simbólicas
- Inclusión de Archivos
- Macros de Compilador
- Compilación Condicional
- Comandos y funciones definidas por el usuario

4.3.1 Constantes simbólicas

Las constantes son aquellos elementos que nunca cambian de valor, por ejemplo:

```
#define ESCAPE 27
WHILE (tecla <>ESCAPE)
        tecla=INKEY (0)
END
RETURN
```

Para definir una constante, se utiliza la directiva #define, por ejemplo:

```
#define FLECHARRIBA  5
#define FLECHABAJO  24
```

Cuando el preprocesador encuentra estos directivos en el programa fuente, remplaza los simbolos en todos aquellos lugares en donde los

encuentre.

Los símbolos definidos son Case-Sensitive (no toman por igual a mayúsculas y minúsculas). La longitud de éstos no tiene restricciones, a diferencia de las variables, que tienen como máximo 10 caracteres para el nombre.

4.3.2 Inclusión de Archivos

La inclusión de archivos es una técnica elegante en el desarrollo de aplicaciones. Esta técnica se basa en los archivos de cabecera.

En la medida en que usted vaya utilizando directivas del preprocesador, la lista crece. Afortunadamente, puede grabar estos mandatos en archivos separados (archivos de cabecera) y llamarlos desde el programa fuente mediante la directiva #include.

4.3.3 Macros del compilador

Una macro de compilador es una funcion insertada directamente en el programa fuente, es decir, cuando el preprocesador encuentra una macro del compilador, no la ejecuta, simplemente la reemplaza en el código fuente en donde se localiza la referencia a dicha macro.

La directiva que se utiliza para definir una macro es #define. Por ejemplo:

```
#define MAXNUMERO(a,b) If( a >b,a,b )
```

de tal forma que se puede referenciar a dicha macro como en el siguiente ejemplo:

```
...
numerox := 3
numeroy := 5
mayor := MAXNUMERO( numerox,numeroy)
```

4.3.4 Compilación condicional

La compilacion condicional permite incuir o excluir código basado en una condicion. Uno de los usos mas frecuentes de la compilacion condicional, se utiliza para el manejo de versiones de una misma aplicación.

Las directivas que se usan para la compilación condicional son "#ifdef" y "#ifndef". Un ejemplo de su uso es el siguiente:

```
#include "inkey.ch"
#ifdef versiondemo
        #define ciclos 2
#else
        #define ciclos 20
#endIf
contador :=1
.
.
.
For contador =1 to ciclos
      .
      .
      .
Next
Return
```

En este ejemplo se usa el identificador versiondemo junto con la constante simbólica ciclos para definir cuantos ciclos se harán durante la ejecución del programa.

4.3.5 Comandos y funciones definidos por el usuario

Se pueden definir comandos y funciones a través de #command y #translate. Una función creada mediante ésta técnica ocupa menos espacio en memoria y es más rápida que una función definida por el usuario (UDF) en forma tradicional.

Se deben tomar las siguientes precauciones para evitar que haya errores en tiempo de compilación ó en tiempo de ejecución:

1.- El nombre de la función es Case-Sensitive.
2.- El número de parámetros de la función debe ser igual al número de parámetros especificados en la línea de definición.
3.- No puede existir espacio en blanco entre el nombre de la función y el paréntesis que abre la lista de parámetros.

A continuación se muestra un ejemplo de una función que tiene como propósito centrar una cadena. Los parámetros de entrada son el número de renglón de la pantalla y la cadena.

```
#translate centra(<ren>,<cad>)=>@<ren>,INT((80-LEN(<cad>))/2);
SAY <cad>
```

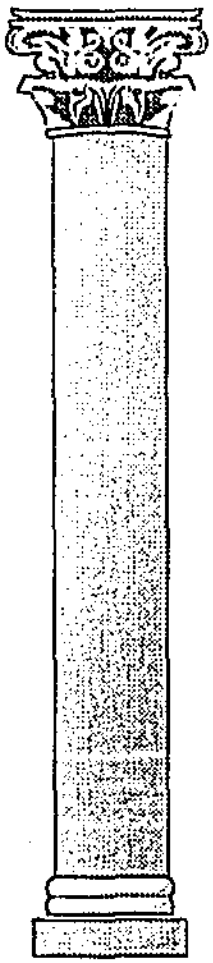
El ejemplo que se muestra a continuación nos da una idea de como se pueden definir comandos en español. El mandato limpia será substituído durante el preproceso por el comando CLEAR.

```
#command limpia => CLEAR
```

Ambos ejemplos se referencian en el siguiente programa:

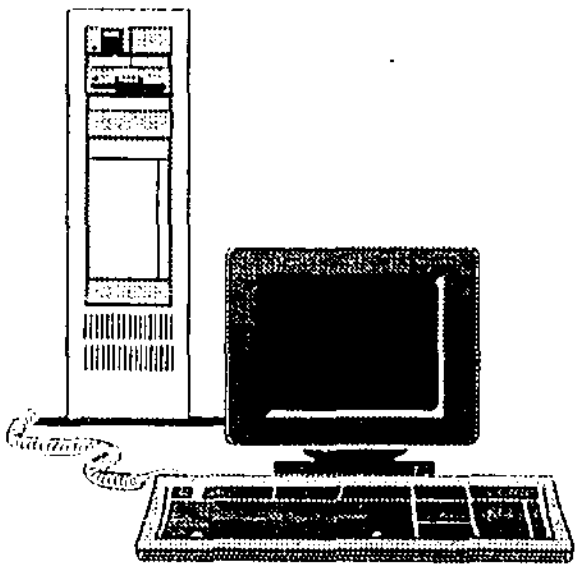
```
/* Programa:          PRUEBA41.PRG      Hecho en Clipper 5.01
   Creación:          01/10/93          Por:   Jorge Castaño
   Ultima revisión:   05/10/93          Por:   Fernando Tecuapacho
    ren:              Número de renglón
    cad:              Cadena                  */
   Propósito:         Ejemplificar el uso de el preprocesador de Clipper   */

#command limpia => CLEAR
#translate centra(<ren>,<cad>)=>@<ren>,INT((80-LEN(<cad>))/2); SAY <cad>

B:=8
limpia
cadena :="ESTA ES UNA CADENA"
centra(B,cadena)
return
```

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO



CAPITULO 5

- PROCEDIMIENTOS Y FUNCIONES DEFINIDOS POR EL USUARIO



ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES
A R A G O N

5.1 VENTAJAS Y APLICACIONES DE FUNCIONES DE USUARIO

Las Funciones definidas por el usuario abren un mundo nuevo para quienes trabajen en un entorno de programación basado en la sintaxis básica de la programación Xbase. Las Funciones definidas por el usuario son un recurso poderoso que Clipper ofrece y permite ahorrar muchas líneas de código. Como beneficios del uso de las funciones definidas por el usuario podemos citar:

- Creación de bibliotecas de funciones que pueden ser usadas por cualquier sistema desarrollado en Clipper.
- Mayor seguridad en la constancia de datos gracias a la utilización de las mismas funciones en todo el sistema.
- Menor trabajo de mantenimiento del sistema.
- Reducción de código redundante.
- Estímulo del uso de la programación estructurada, lo que facilita la construcción y mantenimiento de sistemas.

Aplicaciones principales:

- Centralización de mensajes.
- Rutinas de confirmación del tipo Si/No.
- Creación de pantallas.
- Creación de menús desplegables y otros efectos visuales en la pantalla.
- Funciones generadoras de gráficas.
- Uso en validación de datos, mejorando en mucho la eficiencia de las órdenes *RANGE y VALIDE.*
- Creación de funciones inexistentes en Clipper, por ejemplo: funciones financieras, trigonométricas, funciones de bajo nivel, etc.

Las funciones definidas por el usuario permiten el desarrollo de aplicaciones, las cuales serian muy complejas si no se utilizarán las mismas. En otras palabras, si Clipper no lo tiene, hágalo usted mismo.

Esta arquitectura abierta ha sido una de las principales razones del éxito de Clipper.

5.2 FUNCIONES Y PROCEDIMIENTOS.

¿Una función es lo mismo que un procedimiento?

La respuesta en un principio es afirmativa, pero haciendo algunas matizaciones; una función siempre retorna un valor, mientras que un procedimiento no. Algunas funciones retornan siempre el mismo valor, como por ejemplo: la función que retorna el valor de PI, mientras que otras evalúan y procesan alguna información, pudiendo retornar valores diferentes.

Una función puede retornar cualquier dato válido para Clipper, tal como una cadena de caracteres, número, fecha, valor lógico, etc.

Una función, en Clipper, no necesita ser asignada a una variable. Esto quiere decir que se puede comenzar la linea de una instrucción directamente con una función. Por ejemplo:

```
LOCAL c_color:= [w+/b+]
SETCURSOR(0)
CLEAR
Zooms(10,10,15,30, [w+/b+])
```

Las anteriores líneas de un programa primero limpian la pantalla y después ejecuta la función *Zooms()*.

Componentes de una función y de un procedimiento de usuario:

La estructura para definir una función es la siguiente:

```
[STATIC] FUNCTION <f_nom> [<Lpar>)]
                [LOCAL  <vmem>  [:= <exp>,...]
                [STATIC <vmem>  [:= <exp>,...]
                [FIELD   <lcamp>  [IN  <alias>]
                [MEMVAR <lvar>]
 RETURN (<exp>)
```

<f_nom> Es el nombre que se le da a la función. No se deben utilizar palabras reservadas y no debe exceder de 10 caracteres.

<Lpar> Es la lista de parámetros a usar.

<exp> Es el valor que retorna la función.

LOCAL Declara e inicializa una lista de variables o arrays cuya visibilidad y ciclo de vida estará restringido a esta función y a las que ésta llame.

STATIC Declara e inicializa una lista de variables de memoria que serán visibles sólo en esta función, pero cuyo ciclo de vida será durante toda la aplicación.

MEMVAR Declara una lista de identificadores para usar como variables de memoria privadas o públicas siempre que se encuentren.

STATIC FUNCTION: Determina que esta función de usuario solamente pueda ser llamada por procedimientos y funciones declarados en ese mismo archivo (arch.prg). Esto hace posible que podamos emplear una función con el mismo nombre y un uso distinto en otros programas.

La estructura para definir un procedimiento:

```
[STATIC] PROCEDURE <f_nom> [<Lpar>)]
            [LOCAL  <vmem>  [:= <exp>,...]
            [STATIC <vmem>  [:= <exp>,...]
            [FIELD   <lcamp>  [IN  <alias>]
         RETURN
```

<f_nom> Es el nombre que se le va asignar al procedimiento. No se deben utilizar palabras reservadas y no debe exceder de 10 caracteres.

<Lpar> Es la lista de parámetros que serán pasados. Estos parámetros o variables son declaradas como locales.

LOCAL Declara e inicializa una lista de variables cuya visibilidad y ciclo de vida estará restringido a este procedimiento y a las que éste llame.

STATIC Declara e inicializa una lista de variables de memoria que serán visibles sólo en este procedimiento, pero cuyo ciclo de vida será el de toda la aplicación.

MEMVAR Declara una lista de identificadores para usar como variables de memoria privadas o públicas siempre que se encuentren.

STATIC PROCERDURE: Determina que ese procedimiento solamente puede ser llamado por procedimientos y funciones declarados en ese mismo programa (archivo <arch.prg>). Ello implica que podamos usar el mismo nombre para llamar a dos funciones distintas dentro de una misma aplicación.

5.3 PARAMETROS

En la versión de Clipper 5.01 se pueden especificar parámetros en dos lugares. El primero de ellos en la línea donde está el comando FUNCTION y después el nombre de la función, encerrando los parámetros entre paréntesis y separados por comas. Ejemplo:

```
c_mensaje:= [La impresora no se encuentra preparada...]
n_ren:= 20
n_sonido:= 1
Nota(c_mensaje, n_ren, sonido)


FUNCTION Nota(rot, ren, soni)
/*Autor:      Fernando T. Texis
  Fecha:      Ene-1992
  Objetivo:  Despliega un mensaje centrándolo.
  Entrada:
            rot:   Tipo carácter
            ren:   Tipo numérico
            soni:  Tipo numérico
  El sonido es opcional
  No retorna ningún valor.
*/
 LOCAL pos
 pos:= (80 - LEN(rot))/2
 @ ren, pos SAY rot
 IF PCOUNT() == 3
   TONE(185,4)
   TONE(370,3)
   INKEY(0)
 END
 RETURN(Nil)
```

En este ejemplo, los parámetros se consideran variables locales, y no es necesario especificar la orden PARAMETERS.
Una segunda forma de especificar los parámetros es mediante la orden

PARAMETERS para que los datos pasados a una función sean recibidos y asignados a variables. Si una función no necesita parámetros, la orden PARAMETERS deberá omitirse. Por ejemplo la función que retorna el valor de PI podrá ser llamada simplemente por PI(), sin especificar argumentos dentro de los paréntesis, ya que dicha función ejecuta una tarea especifica, y cualquier valor eventualmente suministrado no deberá alterar su valor.

Si una función crea o utiliza variables durante su procesamiento, procure declararlas como locales, estáticas o privadas así se tendrá más control sobre las mismas. Hacer locales las variables utilizadas internamente es importante, pues si se utiliza fuera de la función puede se modificado, pudiendo dar lugar a un error de ejecución o de resultados.

Parámetros

Los parámetros sólo son variables de memoria usadas para recibir los datos o argumentos pasados a una función o procedimiento. Saber especificarlos, y sobre todo, controlarlos, es fundamental para el funcionamiento de la función y del sistema en general. Ejemplo:

```
n_ren:= 20
n_col:= 20
c_mensaje:= [No se encuentra la clave...]
IF .NOT. FOUND()
   Muevelzq(n_ren, n_col, c_mensaje)
END

FUNCTION Muevelzq(lin1,col1,texto)
/*Muevelzq(lin1 ,col1, texto)
 Autor: 111 Funciones, Ramalho
 Fecha: 1992
 Desplaza el texto a partir de la coordenada especificada de derecha a  izquierda,
 causando una impresión de movimiento del mismo.
 Entrada:
          lin1,col1:    Son parámetros de tipo numériço y representan el renglón y
                        columna de donde parte el texto.
```

```
            texto:      Este parámetro es de tipo carácter.
            Utiliza la función Zoom() que no es propia de Clipper.
SALIDA: No retorna nigua valor.
*/
LOCAL curstat, tv, parada
curstat:=SETCURSOR()
SETCURSOR(0)
IF lin1==NIL
  lin1:=22
END
IF col1==NIL
  col1:=20
END
IF texto==NIL
  texto:="PULSE UNA TECLA PARA CONTINUAR... "
ELSE
  texto:= texto + SPACE(1)
ENDIF
c_color:= SETCOLOR()
TV=SAVESCREEN(lin1-1,col1-1,lin1+1,col1+LEN(texto))
Zoom(lin1-1,col1-1,lin1+1,col1+LEN(texto),c_color,{doble})
parada:=0
WHILE parada==0
  parada=INKEY(.1)                    // Controla la velocidad
  @ lin1,col1 SAY SUBSTR(texto,1,60)
  texto=SUBSTR(texto,2,60)+SUBSTR(texto,1,1)
END
RESTSCREEN(lin1-1,col1-1,lin1+1,col1+LEN(texto),TV)
SETCURSOR(curstat)
RETURN(NIL)
```

En el ejemplo anterior, las tres variables usadas como argumentos de la función (n_ren, n_col, c_mensaje) son recibidas y asignadas a las variables (lin1 ,col1, texto). En este caso, la función se ejecutará correctamente.

Ahora suponga que ocurriera lo siguiente:

```
Muevelzq(n_ren, n_col)
```

Aquí se ha omitido el último parámetro de la función, lo que producirá la interrupción de la ejecución de la función, ya que asignará los dos primeros parámetros a las variables de *n_ren* y *n_col*, y al no haber un tercer argumento la variable *c_mensaje* quedará indefinida, o sea, con el valor NIL, causando un error en el programa.

En el ejemplo siguiente se puede ver otra situación:

```
Mueve_Izq(n_ren, n_col, c_mensaje, 15)
```

Los tres argumentos son asignados correctamente a las variables *n_ren*, n_col y c_mensaje respectivamente, mientras que el cuarto parámetro *<15>* no será tomado en cuenta. En este caso, aun que los parámetros se han especificado incorrectamente, no se producirá ningún error.

Otra situación que puede dar lugar a un error es la asignación de un contenido incorrecto a alguna de las variables. Por ejemplo *n_ren:= [20]* producirá un error de ("type mismatch") durante la ejecución, pues "n_ren" debe de ser variable de tipo numérico.

Comprobación de parámetros

La forma más segura de trabajar con funciones y asegurarse de que no se producirán errores por falta de parámetros o por pasarlos incorrectamente es determinar la cantidad y tipo. Esto puede requerir algún trabajo extra de codificación, pero garantiza que la función estará a prueba de usuarios distraídos o bien de otros programadores o analistas que estén usando la función en otros sistemas.

El secreto para estas comprobaciones son las funciones PCOUNT() y VALTYPE(). Y las comprobaciones que deben hacerse son las siguientes:

- Verificar la cantidad de parámetros y asegurarnos de que todos las variables sean de tipo correcto.

- En algunas ocasiones un parámetros puede ser opcional. En ese caso, se prueba su existencia y, eventualmente, asignarle un valor por default.

- En la versión de Clipper 5.01, como una variable no definida, recibe el valor de NIL, la prueba se puede hacer simplemente de la siguiente forma:

```
FUNCTION Nota(rot, ren, soni)
/*Autor: Fernando T. Texis
 Fecha:      Ene-1992
 Objetivo:  Despliega un mensaje centrándolo en la pantalla.
 Entrada:  rot:   Tipo carácter
           ren:  Tipo numérico
           soni: Tipo numérico
 El sonido es opcional
 No retorna ningún valor.*/

LOCAL pos
IF soni== NIL
  soni:= 1
END
pos := (80 - LEN(rot))/2
@ ren, pos SAY rot
IF PCOUNT() == 3
  TONE(185*soni,4)
  TONE(370*soni,3)
END
RETURN(Nil)
```

Parámetros por referencia y por valor

Un argumento (parámetro) puede pasarse a una función de dos formas: por REFERENCIA o por VALOR.

Cuando un argumento se pasa por valor (modo de default para una función), su contenido original no se altera, ya que la función copia el contenido de la variable pasada en una variable local (dentro de la función), siendo esta última la procesada por la función.

Cuando se pasa una variable por referencia a una función, lo que se envía no es su contenido, si no su dirección de memoria. En otras palabras, no se creará una variable local dentro de la función, si no que se creará un apellido para la variable usada como argumento, siendo la propia variable pasada la que sufrirá las alteraciones.

Para indicar que una variable se pasa por referencia, el nombre de la variable debe ir precedido por el signo "@" (arroba), ejemplo:

```
Local n_m:= 50
? Cuadrado(n_m)      //Argumento pasado por valor; retorna 2500
? n_m                //Retorna 50
?
? Cuadrado(@n_m)     //Argumento pasado por referencia; retorna 2500
?
? n_m                //Retorna 2500
```

Hay que hacer algunas observaciones en relación al paso de parámetros. Los campos de archivos se pasan siempre por valor a las funciones, y deben ir entre paréntesis. Los nombres de las matrices se pasan siempre por referencia. Los elementos de una matriz se pasan por valor.

Cuando se trabaje con procedimientos, asuma el concepto inverso al presentado aquí por las funciones. En los procedimientos, los argumentos se pasan normalmente por referencia. Para pasar un argumento por valor, use el signo arroba "@".

5.4 REGLAS PARA EL DESARROLLO DE FUNCIONES

Para el desarrollo de los programas, es importante considerar un modelo que en términos generales será la forma adecuada de crear aplicaciones.

REGLA No. 1 Nunca desarrolle procedimientos. Desarrolle únicamente funciones. Recordemos que una función siempre regresa un valor. En el caso de una función que no nos interese el valor que regrese, utilice como valor de regreso el nuevo tipo de dato: NIL.

REGLA No. 2 El programa principal deberá iniciar con una función (de acuerdo a la regla No. 1).

REGLA No. 3 Desarrolle funciones parametrizadas.

REGLA No. 4 No desarrolle funciones demasiado largas.

Adicionalmente, puede considerar los siguientes elementos como ayuda:

1.- Indique como prefijo en la variables, si es local ó estática. Además indique el tipo de variable, por ejemplo "n" para numérica, "c" para carácter, etc.

2.- En el estatuto "DO WHILE..." puede omitir el "DO".

3.- En los estatutos "ENDDO" ó "ENDIF" puede especifica unicamente "END".

4.- Utilice como operador de asignación ":=" en lugar de "=".

5.- Utilice como operador de comparación "==" y no "=".

5.5 BIBLIOTECA DE FUNCIONES

Las funciones de usuario se pueden usar de varias maneras:

- Incorporando el listado fuente en el programa escribiendo directamente la función dentro de su programa, preferiblemente al final del mismo, o bien escribir todas las funciones en archivo usando un procesador de textos de código ASCII, que permita copiar bloques.

  La ventaja de este método es que el programa ejecutable tendrá un tamaño menor, pues sólo se incorporan las funciones deseadas.

- Creando un programa .OBJ con las funciones. En este caso, se captarán las funciones y se separan en grupos de utilidad, en seguida se compila cada uno de los programas fuentes:

  Ejemplo:

```
C:\> CLIPPER Fun_varias
```

  Esto genera un archivo llamado "Fun_varias.obj". A continuación tiene que enlazar este programa fun_varias con el suyo:

```
C:\>RTLINK FI menuprinc + fun_varias LIB CLIPPER + EXTEND
```

La ventaja de este método es que una vez compilado el programa que contiene las funciones, basta enlazarlo con el nuestro, evitando así teclear o copiar las funciones de un archivo de textos a nuestro programa. Sin embargo, el tamaño del programa ejecutable será un poco mayor.

Este último método es interesante cuando se tiene funciones en un único archivo. No obstante, si se desarrollan funciones de varios tipos y se crean varios .obj y se necesita enlazar uno o mas de esos archivos a la aplicación, existe una solución mas optima, que a continuación se explica.

5.4 EL MANEJADOR DE LIBRERIAS LIB

Una de las características mas importantes de C, es la facilidad con la que se pueden construir y utilizar librerías de funciones .

Una librería es una colección de funciones precompiladas y agrupadas bajo un mismo nombre (archivo), que pueden ser utilizados por cualquier programa. La ventaja de utilizar tales librerías está en no tener que escribir el código fuente de la función de librería correspondiente en el programa y en el tiempo que se ahorra, ya que el compilador cuando compila nuestro programa, no tiene que compilar tales funciones.

El lenguaje C proporciona el manejador de librerías LIB que viene en los productos de Microsoft, el cual permite crear o mantener librerías (.LIB).

Una librería stand-aline está hecha a base de módulos objeto, esto es, archivos objeto que han sido combinados para formar librería. A diferencia de un archivo objeto, un módulo objeto no existe independientemente de la librería a la que pertenece, y no tiene un camino o extensión asociándose a su nombre.

Cuando enlace la librería (archivo .lib) con su aplicación sólo las funciones necesarias serán incorporadas a su sistema. Si observa el tamaño del archivo CLIPPER.LIB, comprobará que es mayor a 300 kb y el tamaño de su programa .EXE tiene un tamaño mucho menor, esto quiere decir que solamente una parte de la librería CLIPPER.LIB ha sido añadida a su archivo ejecutable.

Para trabajar con el manejador de librerías LIB, se debe de crear un (.OBJ) para cada función que desee integrar a la librería. Esto facilitará el mantenimiento posterior del archivo, además de disminuir el tamaño de su archivo ejecutable, pues solamente las funciones llamadas serán incorporadas.

Utilizando el manejador de librerías LIB, se puede:

- Combinar ficheros objetos para crear una nueva librería
- Añadir ficheros objetos a una librería existente
- Borrar o reemplazar los módulos de una librería existente
- Extraer módulos objetos de una librería existente y colocarlos como ficheros objetos independientes.
- Combinar el contenido de las librerías existentes para formar una única librería.

Sintaxis del programa LIB

LIB (arch_lib) (opciones) (ordenes) , (arch_list) , (arch_salida))).

Arch_lib Indica el nombre de la librería que se desea cambiar o modificar, si la librería no existe, LIB pregunta si se quiere crear. Por defecto, se asume la extensión (.LIB).

(;) Si se escribe un punto y coma (;) a continuación del nombre de la librería, LIB ejecuta solamente un chequeo sobre la librería especificada, este chequeo normalmente no es necesario, ya que LIB verifica cada fichero objeto antes de añadirlo a la librería.

Opciones: el manejador de librería tiene una opción para configurar el tamaño de pagina:

/PACE SIZE: número

La opción /PACE SIZE: especifica el tamaño de la página para la librería.

Número: es una potencia entera de 2 cuyo valor está comprendido en el rango de 16 a 32768. Por defecto el tamaño de la página es de 16 bytes. El tamaño de la página afecta a la alineación de los módulos en la librería.

Los módulos en la librería son siempre alineados para comenzar en una posición y sea un múltiplo del tamaño de la página.

Las ordenes pueden ser cualquiera de las siguientes:

+{fichero-objeto/lib}	Si se especifica un fichero objeto éste es añadido a la librería, su contenido es añadido a la librería existente (archhst). El nombre de la librería debe tener la extensión (.LIB)
-Módulo	Borra el módulo especificado de la librería
-+Módulo	Reemplaza el módulo especificado en la librería, borrando el módulo y sustituyendo por el fichero objeto del mismo nombre.
*Módulo	Copia el módulo especificado, a un fichero objeto.
Arch_list	Es el nombre del archivo de referencias cruzadas. Si no se especifica no se crea. Este archivo contiene una lista alfabética de todos los símbolos globales (public) en la librería y una lista de los módulos de la librería.
Arch_salida	Es el nombre de la librería modificada que LIB crea como salida. Si no se especifica, la librería original es salvada con el mismo nombre y extensión .BAK, permaneciendo con el mismo nombre la modificada.

Ejemplos:

No. 1 Reemplaza el módulo fun-arca en la librería.

```
Libcl15.lib por el archivo objeto del mismo nombre.
LIB libCL15 -+ fun_area;
```

No. 2 En el siguiente ejemplo marca el módulo fun_area de la librería LIBCL15.LIB a un fichero objeto denominado fun-area, con lo que este módulo desaparece de la librería. Y copia el módulo func-vol a un archivo objeto denominado func-vol.ob. La libreria modificada es llamada LIBCL15V.LIB permaneciendo sin cambiar la libreria Llibcli5.LIB.

```
LIB libcli5 -*fun-area *fun-vol, libcli5v;
```

Una segunda forma de crear una librería es ejecutar sin parámetros:

```
LIB
```

y responder a las siguientes preguntas que se muestran en la pantalla:

```
Library Name:
Operations:
Lis File:
Output library:
```

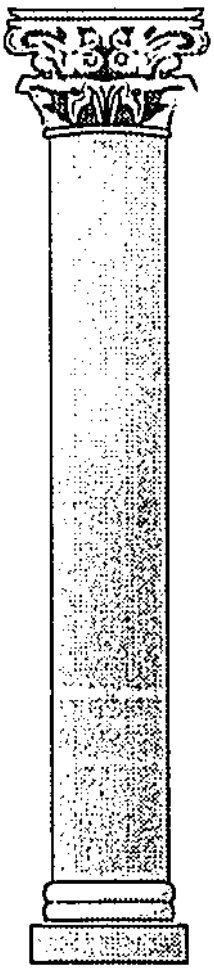
Igual que se indicó para la orden RTLINK, para la orden LIB se puede crear un archivo de respuesta automática, especificando la siguiente orden.

```
LIB C Nombre-archivo.
```

Finalmente para este capitulo se hacen algunas sugerencias para presentar las funciones a los usuarios o a los programadores:

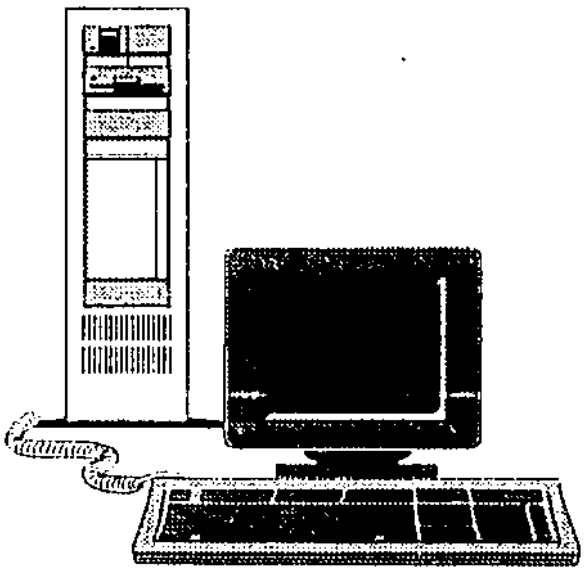
1. Nombre	Nombre que se le asigna a la función.
2. Objetivo	Describe brevemente la función.
3. Entrada	Especifica qué tipo de variables se usan como argumentos.
4. Salida	Especifica el tipo de variables que retorna la función (si fuera el caso).
5. Sintaxis	Presenta la sintaxis de la función.
6. Dependencia	Indica si una función hace uso de otra función que no sea de Clipper.
7. Ejemplos	Cuando es oportuno, se presentan ejemplos de la función.

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO



CAPITULO 6

- ARREGLOS



ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES
A R A G O N

6.1 DEFINCION DE ARREGLOS Y METODOS DE INICIALIZACION

Los arreglos son un tipo de dato y una herramienta importante en la programación, ya que ellos son un modelo que representa algo real, por ejemplo un arreglo de una dimensión son los canales de una televisión, la numeración que sigue una calle, una palabra es un arreglo de letras en donde cada letra representa un elemento del arreglo

Podemos encontrar arreglos de más de una dimensión, así por ejemplo el tablero de un juego de ajedrez es un arreglo de dos dimensiones, los ejes de coordenada en "x" y la coordenada en "y", estas coordenadas son un arreglo de dos dimensiones, si queremos representar un sistema en el espacio, requerimos de tres coordenadas "x", "y" y "z" que nos representan un arreglo de tres dimensiones.

Ahora bien, si en la versión anterior de Clipper quisiéramos representar un arreglo de dos dimensiones, tendríamos que realizar alguna rutina que lo pudiera simular, veamos un ejemplo en concreto.

```
DECLARE arreglo1[2]
DECLARE arreglo1[2]
DECLARE arreglo1[2]

arreglo1[1]= "arreglo2"
arreglo1[2]= "arreglo3"

arreglo2[1]= 10
arreglo2[2]= 20

arreglo3[1]= 30
arreglo3[2]= 40
```

El elemento 1 del arreglo1 nos indica que tenemos que consultar al arreglo2, pero el arreglo2 contiene dos elementos, entonces el elemento 1 del arreglo1 está relacionado con dos elementos que pertenecen al arreglo2, la forma de representar lo anterior es:

- Elemento 1 del arreglo1 con el elemento 1 del arreglo2 (10)
- Elemento 1 del arreglo1 con el elemento 2 del arreglo2 (20)

De la misma forma el elemento 2 del arreglo1, está relacionado con el arreglo3:

- Elemento 2 del arreglo1 con el elemento 1 del arreglo3 (30)
- Elemento 2 del arreglo1 con el elemento 2 del arreglo3 (40)

de lo anterior podemos concluir:

	1	2
1	10	20
2	30	40

De tal forma que simulamos un arreglo de dos dimensiones utilizando arreglos de una dimensión, ahora bien analizando el concepto de simulación, lo que realizamos fue hacer una referencia de un arreglo a otro arreglo, o dicho en otras palabras un elemento apunta a otra serie de elementos, de esta misma forma en Clipper utilizar el concepto de apuntadores para realizar arreglos multidimensionales.

Otros puntos importantes acerca de los arreglos son:

- Los arreglos en Clipper tienen como base el número uno; otros lenguajes como en "C", la base es cero, es decir en Clipper no existe el elemento "0,0". El primer elemento es el "1,1" (en un arreglo de dos dimensiones.

- Cuando se declara un arreglo será del tipo PRIVATE, aunque puede tomar diferentes tipos cuando es definida la variable con anterioridad.

- Los arreglos de una dimensión pueden tener hasta 4096 elementos, recuerde que un elemento de un arreglo puede contener otro arreglo.

- Los elementos de un arreglo pueden contener diferentes tipos de datos.

En Clipper 5.01, la forma más sencilla de declarar un arreglo consiste en especificar la dimensión que se desea manejar, por ejemplo:

```
FUNCTION Declara()
 LOCAL arreglo[5]
 arreglo[1]:= "arreglo3"
 arreglo[2]:= 10
 arreglo[3]:= 20
 arreglo[5]:= 40

 ?arreglo[4]          //Nil
RETURN(NIL)
```

Existen funciones en Clipper para definir arreglos como: ARRAY(), DIRECTORY(), etc., las cuales se explicarán mas adelante.

En Clipper 5.01 sigue existiendo el concepto de arreglo de una sola dimensión, la diferencia es que ahora un elemento de un arreglo puede contener un apuntador a otro arreglo de una dimensión; con este concepto podemos tener arreglos de una dimensión que simulen arreglos de más de una dimensión. Este concepto de simulación de ninguna manera es una matriz, pero su concepto nos ayuda a comprender la utilización de los arreglos.

Existen cantidades físicas y geométricas que no pueden escribirse mediante un solo número, debido a que para su completa identificación se requiere de una dirección así como de una magnitud: si deseamos expresar una columna de una matriz, no podemos hacerlo sólo con un número, lo que nos lleva al concepto de vector, entonces en una matriz existen dos tipos de vectores, los vectores columna y los vectores renglón.

Estos vectores son los que llamamos en Clipper arreglos, entonces un arreglo o vector es un conjunto de números y son útiles por que nos permiten considerar un conjunto de números como un solo objeto, denotado mediante un solo símbolo y hacer cálculos con estos símbolos en poco espacio. La representación matemática así obtenida es muy elegante y útil.

6.1.1 Inicialización de Arreglos

Existen varios métodos para inicializar un arreglo, de los cuales tres son de Summer`87 y son los siguientes:

```
DECLARE aArreglo[<valor>]
PUBLIC   aArreglo[<valor>]
PRIVATE  aArreglo[<valor>]
```

Los nuevos métodos de inicialización son:

```
aArreglo:= { }                  // Arreglo literal
aArreglo:= ARRAY(<valor>)
aArreglo:= Udf()                // Función definida por el usuario
```

El primer ejemplo de los nuevos métodos de inicialización, sólo inicializa el apuntador para el arreglo con el nombre de aArreglo, tiene una longitud de 0, y los elementos sólo pueden ser añadidos por medio de las funciones AADD(), ASIZE() o reinicializando el arreglo por otro método.

Los arreglos literales no necesariamente deben de inicializarse, el único propósito de este tipo de inicialización es crear una dirección de memoria para la variable "aArreglo", y evita que tenga un valor NIL que Clipper asigna por default a las variables que desconoce su tipo.

En el segundo ejemplo, la función ARRAY() inicializa el apuntador y crea "slots" de memoria definidos por <valor>, que es el número de elementos del arreglo.

El tercer ejemplo, permite a una función llenar el arreglo, de acuerdo a las especificaciones de dicha función.

A continuación se muestra la utilización de la primera forma de inicialización de arreglos. En Clipper 5.01:

```
LOCAL aArreglo:= {"a", "b", "c", "d"}
```

En Clipper Summer`87:

```
DECLARE aArreglo[4]
aArreglo[1]= "a"
aArreglo[2]= "b"
aArreglo[3]= "c"
aArreglo[4]= "d"
```

Para arreglos de mas de una dimensión:

```
LOCAL aArreglo:={"a", "b", "c", "d", {"1", "2", "3"}}
```

El elemento 5 es un arreglo de tres elementos, podemos observar que la versatilidad de Clipper 5.01 es mayor a la de sólo obtener matrices, ya que como sabemos la configuración de una matriz siempre es rectangular. En Summer`87 no existe forma directa de definir este tipo de configuración (un arreglo dentro de otro arreglo).

Los elementos del arreglo anterior son (tabla 6.1):

Elemento	Valor
1	a
2	b
3	c
4	d
5,1	1
5,2	2
5,3	3

6.1.1 Arreglos anidados

Una de las características fundamentales de Clipper 5.01 es el manejo de arreglos anidados, estos se crean indicando a un elemento de un arreglo definido que el contenido de dicho elemento será otro arreglo, ejemplo:

```
LOCAL lo_aArray[3]
lo_aArray[1]:= 1
lo_aArray[2]:= 2
lo_aArray[3]:= ARRAY(2)          // Formas de direccionamiento válidas
lo_aArray[3,1]:= 31
lo_aArray[3,2]:= 32
?VALTYPE(lo_aArray[1])      // Numérico
?VALTYPE(lo_aArray[2])      // Numérico
?VALTYPE(lo_aArray[3)       // Arreglo
?VALTYPE(lo_aArray[1])      // Numérico
```

6.2 FUNCIONES DE CLIPPER RELACIONADAS CON ARREGLOS

ARRAY()

La función ARRAY() es un sinónimo de declaración que regresa un arreglo sin inicialización, también nos permite crear arreglos multidimensionales. La sintaxis de dicha función es la siguiente:

ARRAY(<expN1>[,<expN2> [,...]])

<expN1> : Número de elementos en la primera dimensión.
<expN2> : Número de elementos en la segunda dimensión.
<,...> : Número de elementos en la dimensión respectiva.

El siguiente ejemplo nos muestra la utilización de la función ARRAY(), así como el concepto de arreglos contenidos en arreglos.

```
/*FUNCION
  Nombre de la función:   Arreglo
  Autor:                  Fernando Tecuapacho
  Fecha de creación:      07-07-93
  Ultima revisión:        Jorge Castaño
  Descripción:            Llena un arreglo de tres dimensiones
  Argumentos:             - * -
  Retorna:                NIL
  Compilar:               Clipper /n/m/w   */
```

```
FUNCTION Arreglo()
 LOCAL aArreglo := ARRAY(3,2,4)
 LOCAL nI, nJ, nK
 LOCAL getlist
 CLEAR
 FOR nI := 1 TO 3
        FOR nJ := 1 TO 2
              FOR nk := 1 TO 4
                    aArreglo[nI,nJ,nK] := ALLTRIM( STR( nI ) )+","+;
                    ALLTRIM( STR( nj ) )+ "," + ALLTRIM( STR( nk ) )
              NEXT
        NEXT
NEXT

? LEN( aArreglo )                  // 3
? LEN( aArreglo[1] )               // 2
? LEN( aArreglo[1,1] )             // 4
? LEN( aArreglo[1,1,1] )           // 5

? VALTYPE(aArreglo )               // Arreglo
? VALTYPE(aArreglo[1] )            // Arreglo
? VALTYPE(aArreglo[1,1] )                // Arreglo
? VALTYPE(aArreglo[1,1,1] )              // Carácter
RETURN(NIL)
```

LEN(aArreglo): Muestra la longitud del arreglo en la dimensión 1.
LEN(aArreglo[1]): Muestra la longitud del arreglo en la segunda dimensión.
LEN(aArreglo): Muestra la longitud del arreglo en la tercera dimensión.
LEN(aArreglo[1,1,1]): Muestra la longitud del elemento 1,1,1

Si verificamos en la función VALTYPE(), encontramos que tenemos 3 arreglos, correspondientes a cada una de las dimensiones previamente definidas.

AADD()/ASIZE()

Las funciones AADD() y ASIZE() permiten modificar el tamaño de un arreglo. La función AADD() añade dinámicamente elementos a un arreglo; mientras que ASIZE() modifica o cambia el tamaño del arreglo al valor deseado.

La sintaxis de estas funciones es la siguiente:

AADD(<arreglo>,<expresión>)
ASIZE(<arreglo>,<número de elementos>)

ambas funciones proporcionan un control poderoso sobre los arreglos, debido a que como podrá recordar, en Summer`87 no existían métodos directos para permitir el crecimiento de un arreglo.

Podemos definir una función para disminuir el tamaño de un arreglo en un elemento, a través de un directivo de preprocesador. Por ejemplo:

```
#Define ARRESHRINK(lo_arr) ASIZE(lo_arr, LEN(lo_arr) -1)
```

ACHOICE()
ACHOICE() sigue ofreciendo una alternativa en cuanto a la listas seleccionables, a través de un menú popup. Existe en Clipper 5.01 por aspectos de compatibilidad con Summer`87, y la sintaxis se define como:

<var> := ACHOICE(<renglón superior>,
<columna izquierda>,
<renglón inferior>,
<columna derecha>,
<arreglo con elementos de tipo carácter>
[, <arreglo de valor lógico> [, <función definida>]])

ACOPY()/ACLONE()
Las funciones ACOPY() Y ACLONE() permiten copiar arreglos. La forma más rápida de copiar arreglos es a través de ACLONE(); duplica arreglos anidados o arreglos multidimensionales, cuya sintaxis es la siguiente:

<arreglo destino> := ACLONE(<arreglo fuente>)

La función ACOPY() es la misma que se manejaba en Summer`87, y existe únicamente por compatibilidad. La diferencia que existe entre ambas funciones consiste en que ACOPY() únicamente reproduce un arreglo a una sola dimensión.

ADEL()
La función ADEL() elimina el elemento especificado de un arreglo, trasladando los elementos una posición arriba (a partir del número de

elemento a eliminar), quedando el último elemento del arreglo indefinido. ADEL() no modifica la longitud del arreglo. La sintaxis se define como:

ADEL(<arreglo>, <número de elemento>)

AFILL()
La forma más rápida de inicializar un arreglo, es a través de AFILL(), sin embargo es importante considerar que no inicializa arreglos anidados. La sintaxis de AFILL() es la siguiente:

AFILL(<arreglo>, <valor> [,<inicial> [,<no. elementos>]])

AINS()
AINS() es básicamente la función opuesta a la función ADEL(); es decir, traslada los elementos de un arreglo una posición hacia abajo, a partir del número de elemento posicionado, quedando este último indefinido. La sintaxis de AINS() es la siguiente:

AINS(<arreglo>, <número de elemento>)

ASCAN()
De la misma forma en la que se efectúan búsquedas en archivos .DBF, es posible efectuar búsquedas en arreglos. Esta búsqueda se realiza a través de la función ASCAN(), la cual evalúa la expresión que se desea localizar en el arreglo, y regresa el número de elemento en el que se ubica dicha expresión (en el caso de exista). Debido a que la búsqueda es secuencial, en arreglos de grandes dimensiones el proceso es lento. La sintaxis es la siguiente:

ASCAN(<arreglo>, <llave> [, <inicial> [, <elementos>]])

ASORT()
La forma más rápida de ordenar un arreglo es por medio de la función ASORT(), que por default, ordena el arreglo en forma ascendente. La sintaxis de dicha función es la siguiente:

ASORT(<arreglo> [, <inicial> [, <elementos> [, <orden>]]])

<orden> corresponde a un parámetro opcional que consiste de un Code Block (los Code Blocks se explican en el siguiente capitulo) en donde se indica la forma en la que debe ordenar al arreglo especificado.

DBSTRUCT()
La función DBSTRUCT() crea un arreglo bidimensional con la estructura del archivo .DBF que se encuentre abierto. La primera dimensión contiene todos los campos definidos en el archivo, y la longitud o número de elementos será igual al número de campos; mientras que la segunda dimensión contiene cuatro elementos que corresponden al nombre del campo, el tipo, la longitud y los decimales. La sintaxis es la siguiente:

<arreglo> := DBSTRUCT()

por ejemplo, la siguiente función lista la estructura de un archivo .DBF recibiendo como parámetro el alias de dicho archivo.

```
/*FUNCION:        List_Structure()
  Autor:          Fernando Tecuapacho
  Fecha de creación: 07-07-93
  Ultima revisión:  Jorge Castaño
  Descripción:    Lista la estructura de la base de datos
  Parámetros:     El nombre o alias de una base de datos
  Retorna:        NIL
  Compilar:       Clipper /n/m/w
*/

#Include "DbStruct.Ch"
FUNCTION List_Structure(lo_alias)
 QOUT([Fied],[Type],[Len],[Dec])
 AEVAL(lo_alias -> (DBSTRUCT()),;
            {|dbf| QOUT(PAD(dbf[DBS_NAME],10),;
             [ ]+ dbf[DBS_TYPE],;
             [ ]+ dbf[DBS_LEN],3),;
             [ ]+ dbf[DBS_DEC],3))})
RETURN(NIL)
```

DIRECTORY()

La función DIRECTORY() crea un arreglo bidimensional, es decir un arreglo de subarreglos, de información de archivos en disco, y en donde cada subarreglo tiene 5 elementos que contiene el nombre del archivo, el tamaño (en byte), la fecha, la hora y el atributo. La sintaxis de esta función es la siguiente:

<arreglo> := ADIRECTORY(<máscara>, <atributo>)

en donde la <máscara> corresponde a la especificación de archivos, por ejemplo "*.DBF", "C:\CLIPPER5*.*", etc.; y por otra parte, el <atributo> corresponde a un carácter que puede tener los siguientes valores y significados:

H - Incluye archivos ocultos
S - Incluye archivos del sistema
D - Incluye directorios
V - Busca el volumen de DOS y excluye el resto.

El programa Dirfuncs.Prg que aparece en el listado muestra un archivo de funciones de manejo de arreglos. Para poder observar la utilización de dichas funciones, ejecutar el programa Direc.Exe utilizando los archivos make.

```
/*Nombre del programa:  Direc.Prg
  Autor:                Fernando Tecuapacho
  Fecha de creación:    07-07-93
  Ultima revisión:      Jorge Castaño
  Descripción:          Carga el directorio en archivo .TXT
  Parámetros:
  Retorna:              NIL
  Compilar:             Clipper /n/m/w
*/

#Xcommand Default <lo_p1> To <lo_v1> [, <lo_p2> To <lo_v2> ] =>;
          <lo_p1> := If( <lo_p1> == Nil, <lo_v1>, <lo_p1> ) ;
        [<lo_p2> := If( <lo_p2> == Nil, <lo_v2>, <lo_p2> ) ]
```

```
Function Direc( lo_path, lo_file, lo_kind )
  Default lo_path To Lb_CurDir() + [\]
  Default lo_file To [Dir.Txt]
  Default lo_kind To [N]
  Set( _SET_PRINTFILE, lo_file )
  Set( _SET_DEVICE, [PRINTER] )

  ListDir( Directory( lo_path, lo_kind ) )
  Set( _SET_DEVICE, [SCREEN] )
  Set( _SET_PRINTFILE, [] )
Return( Nil )

/*Nombre del programa:  DirFunes.Prg
  Autor:                Fernando Tecuapacho
  Fecha de creación:    07-07-93
  Ultima revisión:      Jorge Castaño
  Descripción:          Utilerias para funciones de directorios.
  Parámetros:
  Retorna:              NIL.
  Compilar:             Clipper /n/m/w
*/

#Include  "Directry.Ch"
#Define   LINEFEED  Chr( 13 ) + Chr( 10 )
#Xcommand Default <lo_p1> To <lo_v1> [, <lo_p2> To <lo_v2> ] => ;
          <lo_p1> := If( <lo_p1> == Nil, <lo_v1>, <lo_p1> ) ;
        [; <lo_p2> := If( <lo_p2> == Nil, <lo_v2>, <lo_p2> ) ]

Function Directory( lo_path, lo_kind )
  Local lo_i, lo_name, lo_d_, lo_r_ := {}
  Default lo_path To Lb_CurDir() + [\]
  Default lo_kind To [N]

  lo_d_  := Directory( lo_path + [*.*], [D] )
  If ( lo_kind == [N] )
       [F_NAME] } )Asort( lo_d_,,, { |lo_e1, lo_e2| lo_e1[F_NAME] < lo_e2
  ElseIf ( lo_kind == [D] )
     Asort( lo_d_,,, { |lo_e1, lo_e2| ;
     Dtoc( lo_e1[F_DATE] ) + lo_e1[F_TIME] <;
     Dtoc( lo_e2[F_DATE] ) + lo_e2[F_TIME] })
  Else
    * >>
  End
```

```
  For lo_i := 1 To Len( lo_d_ )
    lo_name := lo_d_[ lo_i, F_NAME ]
    If ( lo_d_[ lo_i, F_ATTR ] == [D] )
      If !( lo_name $ [.,..] )
        Aadd( lo_r_, { lo_name, Directory( lo_path + lo_name + [\], lo_kind ) } )
      End
    Else
      Aadd( lo_r_, lo_name )
    End
  Next lo_i
Return( lo_r_ )


Function ListDir( lo_a_, lo_level )
  Local lo_i
  Default lo_level To 0

  For lo_i := 1 To Len( lo_a_ )
    If ( Valtype( lo_a_[ lo_i ] ) == [A] )
      ListDir( lo_a_[ lo_i ], lo_level + 1 )
    Else
      DevOut( Space( lo_level * 4 ) + Str( lo_i, 4 ) + [: ] )
      DevOut( lo_a_[ lo_i ] + LINEFEED )
    End
  Next
Return( Nil )


/*Make File: Direc.Rmk
  Nombre del programa:   DirFuncs.Prg
  Autor:                 Fernando Tecuapacho
  Fecha de creación:     07-07-93
  Ultima revisión:       Jorge Castaño
  Descripción:           Genera el archivo  Direc.exe
*/

// Define Object's Macro
Objs = Direc.Obj DirFuncs.Obj
// Inference Rule for Compile
.Prg.Obj:
  SAY Obteniendo-->$@
  CLIPPER $* /n/m/w
```

```
// Dependance Statements for .Obj files
Direc.Obj    : Direc.Prg
DirFunes.Obj : DirFunes.Prg
// Dependance Rule for linking
Direc.Exe : $(Objs)
REM Obteniendo-->$@
RTLINK FI Direc,DirFunes LIB \Curclipp\ArLib\Arlib /PIL:Base50
```

AEVAL()

La utilización de la mayoría de las funciones anteriores es relativamente sencilla, por lo cual nos enfocaremos a la función más poderosa AEVAL().

La función AEVAL() permite procesar parte de un arreglo o bien en su totalidad, enviándole como parámetro, el arreglo a procesar y un Code Block par que lo ejecute sobre cada uno de los elementos del arreglo. La sintaxis de esta función es la siguiente:

AEVAL(<arreglo>, <code block> [, <inicial>, <elementos>])

en donde <inicial> corresponde al número de elemento inicial del arreglo que se desea procesar, y <elementos> corresponde al número de elementos que se desea procesar a partir del <inicial>.

AEVAL() se comporta en una forma similar a el ciclo FOR ... NEXT; básicamente AEVAL() requiere de dos elementos como parámetros, un arreglo con el cual trabajar, y un Code Block que le dice qué hacer con cada elemento en el arreglo.

AEVAL() trabajará con cada elemento del arreglo a menos que en su llamada se le especifique un número limitado de los mismos. Veamos el siguiente ejemplo:

```
FUNCTION Uno()
  LOCAL aArreglo:= {}
  LOCAL nI:= 1
  WHILE nI <= 10
       AADD(aArreglo,CHR(64+nI++))
  END
  AEVAL(aArreglo, { | x| QOUT(X) })
RETURN(NIL)
```

La operación que se efectuará en cada uno de los elementos, esta descrita por el Code Block (segundo parámetro), en este caso sólo despliega el valor del elemento en la pantalla.

El arreglo anterior sólo es de una dimensión, tomemos el arreglo de la función Arreglo() que está definida como:

```
LOCAL aArreglo:= ARRAY(3,2,4)
```

Teniendo en cuenta que cada dimensión es un arreglo, podemos deducir que necesitamos de un AEVAL() para cada dimensión, para recorrer la totalidad del arreglo, podemos utilizar el concepto de anidamiento y obtener de esta forma, la siguiente expresión:

```
AEVAL(aArreglo, { |x| AEVAL(x, {|y| AEVAL(y, {|z| QOUT(z) }) }) })
```

ahora bien si tuviéramos 3 arreglos, la forma de evaluar cada uno de ellos sería:

```
AEVAL(aArreglo, { |x| QOUT(x) })
AEVAL(aArreglo, { |y| QOUT(y) })
AEVAL(aArreglo, { |z| QOUT(z) })
```

como sabemos, en este caso "x" es un apuntador a un arreglo al igual que "y", la operación que debemos de utilizar en lugar de QOUT(x) es:

```
AEVAL(aArreglo, { |y| QOUT(y) })
```

y en lugar de QOUT(y) es:

```
AEVAL(aArreglo, { |z| QOUT(z) })
```

haciendo dichas sustituciones llegamos a la expresión anterior o sea:

```
AEVAL(aArreglo, { |x| AEVAL(x, {|y| AEVAL(y, {|z| QOUT(z) }) }) })
```

la cual nos desplegará la totalidad de elementos del arreglo. Esta forma es útil hasta cierto punto, si nos encontramos con arreglos de 10 dimensiones, resultaría muy complicado anidar ese número de AEVAL().

Podemos crear una función recursiva la cual nos detecte cualquier número de dimensiones, y con ello ahorrarnos el trabajo de crear AEVAL() anidados.

6.3 TECNICAS DE MANEJOS DE ARREGLOS

Los arreglos en Clipper 5.01 son una modificación a este tipo de dato con respecto a Summer'87. El manejo adecuado de dichos arreglos debe ser un elemento importante a considerar en la explotación de los mismos.

La estructura que soporta Clipper para el manejo y procesamiento de arreglos puede ser tan compleja como se desee.

El abuso en el manejo de esta estructura puede repercutir en el desgaste de memoria, que finalmente es uno de los principales recursos que debemos cuidar. Debido a que los arreglos anidados representan una estructura de datos poderosa, y considerando el aspecto de memoria, en ocasiones es útil salvar dichos arreglos en disco.

Clipper no soporta el almacenamiento de arreglos en archivos .MEM, de tal suerte que tenemos que desarrollar nuestras propias rutinas. Dichas rutinas pueden resultar complejas debido precisamente a la existencia de los arreglos anidados.

Para poder desarrollar una función que permita grabar el contenido de un arreglo a disco, el primer paso será visitar cada uno de los elementos del arreglo.

Para un arreglo de una dimensión con elementos de tipo simple, el proceso es fácil si pensamos en el estatuto FOR...NEXT; sin embargo ¿qué sucede si un elemento del arreglo es orto arreglo, y así sucesivamente?.

Lo más conveniente para el desarrollo de una función de este tipo es la recursividad.

La recursividad es un atributo que tienen las funciones matemáticas, que consiste en definir una función en términos de si misma.

Ejemplos de programas para inicializar arreglos que contienen más de una dimensión.

```
/*FUNCION
  Nombre:           Lb_afill
  Autor:            Fernando Tecuapacho
  Fecha de creación: 07-07-93
  Ultima revisión:  Jorge Castaño
  Descripción:      Esta función Inicializa  un arreglo con el
                    número cero.
  Argumentos:       <aArray>:  Es el arreglo que se va a inicailizar
  Retorna:          NIL
  Compilar:         Clipper /n/m/w
*/
FUNCTION Lb_afill( aArray )
         LOCAL nJ := 1
         AEVAL(aArray,{|e| IF( VALTYPE( e ) == "A",;
         LB_AFILL( e ),;
         aArray[ nJ++] := 0 ) })
RETURN( NIL )
```

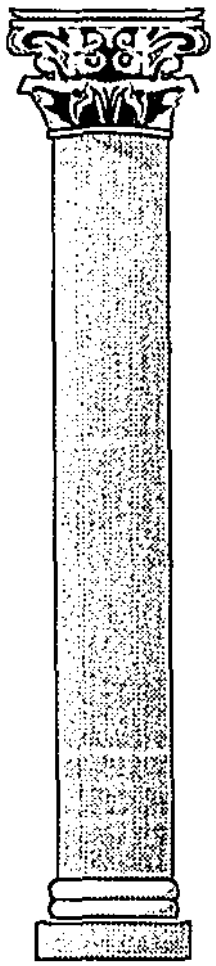
Aún mejor, modifiquemos la función Lb_afill, para enviarle el parámetro con el que se llene nuestro arreglo.

```
/*FUNCION
  Nombre de la función:  Lb_afill
  Autor:                 Fernando Tecuapacho
  Fecha de creación:     07-07-93
  Ultima revisión:       Jorge Castaño
  Descripción:           Esta función inicializa  un arreglo con el
                         elemento que se le especifique.
  Argumentos:            <aArray>:  Es el arreglo que se va a inicailizar.
                         <xfill>    :  Es el elemento con que se va a rellenar
  Retorna:               NIL
  Compilar:              Clipper /n/m/w
*/
```

```
FUNCTION LB_AFILL( aArray, xFill )
        LOCAL nJ := 1

        AEVAL(aArray, {|e| IF( VALTYPE( e ) == "A",;
        LB_AFILL( e,xFill ),;
        aArray[ nJ++] := xFill ) })
RETURN( NIL )
```

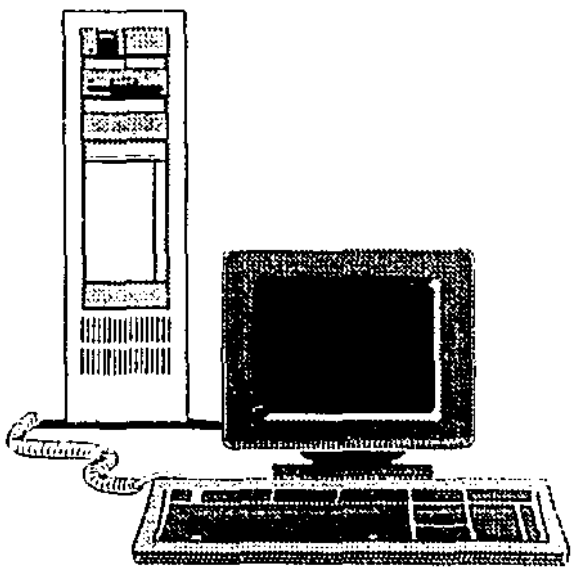
De la misma forma podemos crear varias funciones que nos ayuden en el acceso de información en arreglos multidimensionales.

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO



CAPITULO 7

- CODE BLOCKS Y MACROS



ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES
A R A G O N

7.1 LAS MACROS Y LAS EXPRESIONES EXTENDIDAS

En Clipper 5 existen tres alternativas para crear comandos flexibles que puedan ser usados bajo una gran variedad de situaciones:

Las macros
Las expresiones extendidas
Los code blocks

7.1.1 Que son las macros

Las macros son cadenas de caracteres que son compiladas en tiempo de ejecución e inmediatamente después son ejecutadas. Al igual que dBASE III+, Clipper 5 usa el ampersand (&) para substituir el nombre de una variable por su contenido. Ejemplo:

```
condic:= "cvecli ='B12579'"
USE cliente
LIST cvecli,balance,telef FOR &condic.
/*  Se listarán todos los registros en donde clave sea B12579  */
```

El punto final después del nombre de la variable sirve para indicar a la macro donde termina el nombre de la variable. Este punto no es obligatorio, pero si es recomendable, principalmente si la macro está en medio de una expresión.

Un uso muy común en las macros es en la apertura de archivos:

```
archivo:='costos'
USE &archivo
/*  Se abrirá el archivo costos  */
```

El siguiente ejemplo es un uso inadecuado de las macros

```
USE cliente
campos :="cvecli,balance,telef"
LIST &campos.          &&  El programa no se ejecutará adecuadamente
```

La macro substitución no se realiza adecuadamente cuando la expresión es una lista separada por comas. En el ejemplo anterior sólo se listará el último campo.

7.1.2 Que son las expresiones extendidas

En Clipper 5 también se pueden usar paréntesis en lugar del operador & para realizar la macro substitución. Una expresión extendida es caracterizada por el nombre de la variable entre paréntesis. Ejemplo:

```
defa :="c:\clipper5\programa"
Set default to (defa)
```

Las expresiones extendidas y las macros tienen un uso parecido, pero existe una diferencia importante entre ellas. El valor resultante al evaluar una expresión extendida siempre debe ser de tipo caracter, de lo contrario aparecerá un error en tiempo de ejecución.

7.2 QUE SON LOS CODE BLOCKS

Para poder comprender con mayor claridad los aspectos de optimización, para codificación y ejecución de programas con que cuenta Clipper 5, es importante mencionar los tres nuevos tipos de datos manejados; nulo (NIL), objeto (OBJECT) y bloque de código (CODE BLOCK).

Uno de los aspectos más interesantes de Clipper 5 son los Code Blocks que por cuestiones de abreviación llamaremos de aquí en adelante CB. Los CB permiten soportar conceptos tales como modularidad, encapsulamiento, herencia y programación en objetos.

Como se mencionó anteriormente, un CB es un tipo de dato y por ende, al ser evaluado por la función VALTYPE(), se obtiene una letra "B" correspondiente al tipo de dato CB.

Los CB almacenan expresiones compiladas. Es decir, los CB son traducidos durante la compilación, y no al momento de la ejecución, como ocurre con las macros. Dichas expresiones pueden ser enviadas como parámetros, que en algún momento se evaluarán y generarán un resultado.

Para definir un CB, es necesario declarar una variable y posteriormente asignarle el tipo, que en este caso será el CB, La sintáxis general para la creación de los CB es:

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

La sintaxis extendida es:

{| [<Param 1 > [,<Param 2 >[,...]]] | <Exp 1>[,<Exp 2>[,...]]}

Ejemplo de declaración de un CB

```
miCB := { | radio | radio*radio*3.1416  }
```

Donde radio es la variable de entrada que será elevada al cuadrado y multiplicada por 3.1416. Funcionalmente es equivalente a:

radio*radio*3.1416

Los Code Blocks se pueden evaluar a través de las funciones predefinidas en Clipper: EVAL(), AEVAL() y DBEVAL().

7.2.1 Función EVAL()

Sintáxis: EVAL(<CB>,[<Parámetros>])

Donde CB es el Code Block a ser evaluado y <Parámetros> es la lista de parámetros para evaluar el CB. Por ejemplo:

```
/* Programa:          PRUEBA71.PRG      Hecho en Clipper 5.01
   Creación:          01/10/93          Por:   Jorge Castaño
   Ultima revisión:   05/10/93          Por:   Fernando Tecoapacho

   Propósito:         Ejemplificar el uso de la función EVAL()      */

CLS
miCB := { | radio | radio*radio*3.1416}
QOUT(EVAL(miCB,2))          && Se despliega en pantalla 12.5664

/*      El paso de parámetros también puede ser mediante expresiones,
        por ejemplo:                                                  */

radio1 :=1
radio2 :=2
QOUT(EVAL(miCB,radio1 +radio2))   && Se despliega en pantalla  28.2744
```

7.2.2 Función AEVAL()

Sintaxis: AEVAL(<Arreglo>,<CB>[,<inicio>] [,<número>])

<Arreglo> El arreglo con el que se evaluará el CB.

<CB> El CB a ser evaluado por los elementos del arreglo.

<inicio> El número de elemento con el que se inicia la evaluación del arreglo. Si éste argumento es ómitido, el default es 1.

<número> La cantidad de elementos a ser evaluados. Si éste argumento es omitido, el default es todos los elementos a partir de <inicio>.

Esta función evalua un CB mediante cada uno de los los elementos de un arreglo. El número de elementos usados y el número de elemento donde se inicia son opcionales.

En el siguiente ejemplo, el primer arreglo agrega al segundo aquellos elementos que sean mayores a 7. Posteriormente visualiza todos los elementos del segundo arreglo.

```
/* Programa:        PRUEBA72.PRG     Hecho en Clipper 5.01
   Creación:        06/10/93         Por:  Jorge Castaño
   Ultima revisión: 07/10/93         Por:  Fernando Tecoapacho

   Propósito:       Ejemplificar el uso de la función AEVAL()    */

CLEAR
arreg1:={0,1,2,3,4,5,6,7,8,9}       && Crea un arreglo.
arreg2:={}       && Crea un arreglo vacio.

/*   En el siguiente CB, la función IF() es usada junto con la función AADD
     para agregar en arreg2 aquellos elementos que sean mayores de 7 */

AEVAL(arreg1,{ |elemen| IF(elemen >7,AADD(arreg2,elemen),NIL)})
QOUT(LEN(arreg2))     && Se visualiza un 2.
AEVAL(arreg2, { |elemen| QQOUT(elemen)})

/*      Son visualizados los dos elementos de arreg2 (8 y 9)        */
```

7.2.3 Función DBEVAL ()

Sintáxis:	DBEVAL(<CB>,[<condición FOR],[<condición WHILE>], [<numreg>],[<regesp>],[<plantilla>]
<CB>	El CB a ser evaluado.
<condición FOR>	Es una condición especial que hace que los registros sean procesados, retornando un valor lógico. Cuando el valor retornado es falso, el ciclo termina.
<condición WHILE>	Si se especifica esta condición, serán procesados sólo los registros que satisfagan la condición. Es parecida al comando WHILE.
<numreg>	Especifica el numero de registros que serán procesados a partir del registro actual.
<regesp>	Especifica cierto registro a ser procesado.
<plantilla>	Es un valor lógico que al ser especificado como verdadero, se asume la plantilla REST. En caso contrario se asume la plantilla ALL.

Esta función evalúa un CB para cada registro de un archivo de datos (.DBF). Actúa en el área actual, salvo que sea referida por otra área.

Cada una de las opciones que no sea utilizada, deberá ser dejada en blanco, y su lugar dentro de la sintaxis deberá ser representado por dos comas, sin dejar nada entre ellas.

El siguiente ejemplo muestra el poder de DBEVAL(). Con una sola instrucción se genera un reporte.

```
/* Programa:          PRUEBA73.PRG     Hecho en Clipper 5.01
   Creación:          06/10/93         Por:  Jorge Castaño
   Ultima revisión:   07/10/93         Por:  Fernando Tecoapacho

   Propósito:         Ejemplificar el uso de la función DBEVAL()
```

```
Estructura del archivo cliente:

   cveecli          Clave del cliente
   balance          Balance del cliente
   telef            Teléfono del cliente
   fechapag         Fecha del último pago      */

USE cliente
DBEVAL({ || IF (DATE() - fechapag >30, QOUT(cveecli,balance,telef);
,NIL)};NIL)

/*  Se listarán los renglones correspondientes a aquellos clientes
    que tengan su última fecha de pago superior a 30 dias          */
```

ASCAN ( )	Busca en un arreglo una expresión y regresa el numero del primer elemento que cumple con la búsqueda.
ASORT ( )	Ordena los elementos de un arreglo en forma ascendente ódescendente.
ERRORBLOCK ( )	Crea un nuevo manejador de errores para Code Blocks.
FIELDBLOCK ( )	Regresa un Code Block para obtener ó asignar el valor de un campo de un archivo .DBF
FIELDWBLOCK ( )	Desempeña la misma operación que la función FIELDBLOCK( ), pero ádemas acepta un argumento extra que es usado para especificar el area de trabajo en la cual la operación deberá ser realizada.
MEMVARBLOCK ( )	Crea un Code Block que obtiene ó asigna el valor de una variable cuando el Code Block es evaluado.

Tabla 7.1 Las funciones más comunes relacionadas con Code Blocks

7.2.4 Otras funciones para trabajar con code blocks

Aunque muchas de las nuevas funciones de la librería estándar de 5.01 requieren CB como parte de su sintaxis, sólo tres han sido creadas para expresar trabajo con CB en términos reales: EVAL(), AEVAL() y DBEVAL(). Estas tres funciones fueron tratadas en el punto anterior. En la tabla 7.1 se muestra una lista de funciones que se relacionan con CB.

7.3 VENTAJAS DE LOS CODE BLOCKS SOBRE LAS MACROS

Aunque los CB tienen una fuerte similitud con las macros, existe una significativa diferencia entre ellos. Las macros contienen cadenas de caracteres que son compiladas en tiempo de ejecución e inmediatamente después son ejecutadas. En cambio los CB son compilados en tiempo de compilación, junto con el resto del programa. Por esta razón los CB son más eficientes que las macros ofreciendo la misma flexibilidad y muchas ventajas más.

La diferencia entre macros y CB tiene más relevancia cuando se usan variables declaradas. Dado que la declaración de variables se efectúa en tiempo de compilación al igual que los CB.

A continuación se resumen las ventajas más sobresalientes de los CB sobre las macros:

1) Las macros se ejecutan más lento dado que las macro expansiones deben tomar cada caracter en la expresión e interpretar cuál es su significacia contra la tabla de símbolos.

2) Un CB no es más que la dirección que apunta a una pieza de código compilado o un apuntador a una localidad de memoria a una expresión previamente macro expandida.

3) Los CB ayudan a eliminar conflictos por el manejo de variables, ya que los CB en si mismos pueden ser usados como variables estáticas (STATIC) o locales (LOCAL) extendiendo la visibilidad de las variables usadas por ellos mismos.

4) Los CB pueden ser enviados como parámetros a funciones, permitiendo operar en forma dinámica.

Con el propósito de ilustrar como operan las macro expansiones y los CB, examinemos el siguiente programa:

```
/* Programa:        PRUEBA74.PRG     Hecho en Clipper 5.01
   Creación:        06/10/93         Por:   Jorge Castaño
   Ultima revisión: 12/10/93         Por:   Fernando Tecuapacho
   Propósito:       Ejemplificar las ventajas de los CB sobre las macros
*/

#DEFINE leyenda1  "Llamada a la función:"
#DEFINE leyenda2  "Macro Limitante  ::"
CLS
funcmacro()
funcblock()
/* Fin del cuerpo del programa. A continuación van las funciones  */

FUNCTION funcmacro()
Local arrefun :={"FDsp1()","FDsp2()","FDsp3()"}
Local origen :="Macro Expansión::"
Local origen,evalua
FOR origen:=1 to LEN(arrefun)
      evalua :=&(arrefun[origen])
NEXT
RETURN(NIL)

FUNCTION funcblock()
Local aBlock :={{|| FDsp1(origen)},{|| FDsp2(origen)},{|| FDsp3(origen)}}
Local origen := "Evaluación de Code Block::"
Aeval(aBlock,{|e| Eval(e)})
RETURN(NIL)

FUNCTION FDsp1(origen)
origen:=IF(VALTYPE(origen) =="U",leyenda2,origen)
QOUT(origen+ LEYEND1 + PROCNAME() )
RETURN(NIL)

FUNCTION FDsp2(origen)
origen:=IF(VALTYPE(origen) =="U",leyenda2,origen)
QOUT(origen+ LEYEND1+ PROCNAME() )
RETURN(NIL)

FUNCTION FDsp3(origen)
origen :=IF(VALTYPE(origen) =="U",leyenda2,origen)
QOUT(origen + leyenda1 + PROCNAME() )
RETURN(NIL)
```

En la función funcmacro() se macro expande, uno a uno por medio de un ciclo For...Next, el contenido de los elementos de un arreglo, los cuales representan una serie de funciones a ser ejecutadas.

En la función funcblock() se evalúa un CB, en al cual se contiene la misma lista de funciones contenida en el arreglo de la opción de macro expansión, ejecutándose éstas en forma directa.

Como se podrá observar, el código contenido en la función funcblock() es más óptimo que el contenido de la función funcmacro(), pero no sólo eso, dado que el proceso de macro expansión se efectúa al tiempo de ejecución, éste es más lento. Además, el hecho de usar macro expansión, genera variables dinámicas que fragmentan en gran proporción la memoria.

Se ejecutaron por separado las funciones funcblock() y funcmacro(), en un ciclo For Next desde1 hasta 1000, dando como resultado 54seg para CB y 56 para las macros. Esta diferencia no es importante pero se sabe que la diferencia en el tiempo de ejecución entre macros y CB se acentúa conforme se vayan manejando mayor cantidad de datos.

La prueba se hizo en una IBM PC modelo 70, con procesador 386 a 16 Mhz y 4 Mb de memoria RAM.

7.4. FUNCIONES DE CODE BLOCKS

Una de las técnicas más importantes en el manejo de los CB consiste en utilizarlos en base a funciones que regresen como valor un CB. Por ejemplo si quisieramos definir a la tecla F1 como una tecla "hot key", comúnmente utilizaríamos el comando SET KEY TO de Clipper. Por ejemplo:

```
DEFINE F1 28              //      La tecla F1 es la número 28
SET KEY F1 TO ayuda()
```

Sin embargo durante la ejecución de la función "ayuda()" es importante eliminar las definiciones existentes de algunas teclas, de tal forma que no se deban ejecutar desde el módulo de ayuda.

Para lograr este objetivo, podemos utilizar la función SETKEY() que regresa como valor un CB (si es que existe uno asignado a la tecla en particular) o un valor Nil. De acuerdo al ejemplo anterior, podemos reescribirlo como:

```
DEFINE F1 28    //    La tecla F1 es la número 28
SETKEY(F1, {|procedi,linea,vari|  ayuda(procedi,linea,vari)})
```

En donde:

proced = PROCNAME()
linea = PROCLINE()
vari = READVAR()

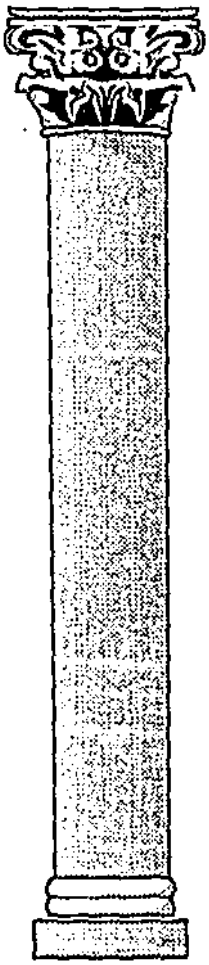
Ahora bien, dentro de la función ayuda() es importante eliminar las definiciones de otras teclas. A continuación se da un ejemplo de la forma en que podría quedar la función ayuda(), pero utilizando solo el nombre del procedimiento a fin de facilitar la explicación.

```
/*      inicio del programa          */
.
.
.
```

```
/*      Se le asigna la función ayuda() a la tecla F1:    */
SETKEY(28, {|procedi| ayuda(procedi)}).
.
.
.
/*      Fin del programa        */


/* Función:          ayuda()       Hecha en Clipper 5.01
   Creación:         19/10/93      Por: Jorge Castaño
   Ultima revisión:  20/10/93      Por: Fernando Tecoapacho

   Propósito:                      Presentar ayuda en pantalla dependiendo
                                   del procedimiento que se este ejecutando

   Esta función necesita un archivo llamado AYUDA.DBF el cual
   debe tener la siguiente estructura:

   campos:   Tipo:  Long.

   PROCE     CAR    8    <- Este es el campo para la indexación
   TEXTO     MEMO                                                    */

FUNCTION ayuda(procedi)
LOCAL pantalla,Flantes,pro
pantalla :=SAVESCREEN(9,44,23,79)  && Se guarda la pantalla actual
/*      Se guarda la asignación actual de la tecla F1 y se le asigna un nulo.    De ser necesario,
las demás teclas de funciones se desactivarían de la misma forma:                 */

Flantes :=SETKEY(28,{|| NIL})
IF procedi ==NIL
   RETURN(NIL)       // Se le asigna a la variable pro el
ELSE                 // el parámetro de entrada si es que
   pro :=procedi     // existe. Si no existe, termina la
ENDIF                // función.
@9,44 TO 23,79
@10,45 CLEAR TO 22,78
@9,59 SAY "AYUDA"
@21,45 SAY REPLICATE(CHR(196),34)
@22,46 SAY CHR(24) +CHR(25) +" <Re Pág><Av Pág>  <Esc> Salir"
USE AYUDA INDEX AYUDA1 ALIAS AYU NEW
SEEK pro
```

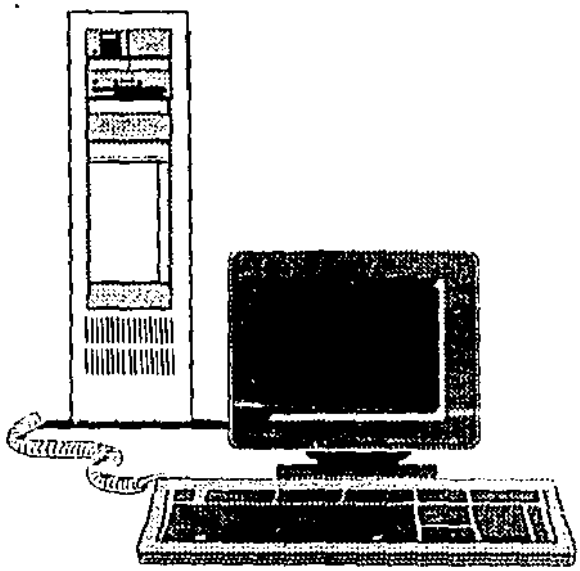
```
IF .NOT. FOUND()
  @12,46 SAY "NO HAY AYUDA DISPONIBLE"
  INKEY(0)
ELSE
  MEMOEDIT(TEXTO,10,45,20,77,,.F.)    //  Se despliega la ayuda
ENDIF
CLOSE AYU
RESTSCREEN(9,44,23,79,pantalla)          //  Se reestablece la pantalla
/*      Se le devuelve a la tecla F1 su anterior asignación:      */
SETKEY(28,Flantes)
RETURN(NIL)
```

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO



CAPITULO 8

- LA CLASE TBROWSE Y TBCOLUMN



ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES
A R A G O N

8.1 CONCEPTOS

La clase TBROWSE contiene objetos que facilitan las operaciones de manipulación de archivos .DBF, al estilo de la función DBEDIT(), pero con muchas más posibilidades.

Permite la presentación y edición de datos en forma de tablas. La recuperación de datos se realiza mediante la evaluación de los CB. Cada objeto TBROWSE depende de los objetos creados por la clase TBCOLUMN que se verá más adelante.

Para poder entender la clase TBROWSE es necesario conocer los siguientes conceptos:

OBJETO:
Es un conjunto de datos que se encuentran encapsulados por el código que opera en ellos. Todo el comportamiento del objeto está descrito en su código y en el estado actual en el que se encuentre dicho objeto. Los datos del objeto están protegidos de otros objetos.

CLASE:
Es un objeto "especial" cuya función consiste en crear objetos en base a un modelo o descripción previamente definido. Podemos decir que una clase es una fábrica de objetos.

INSTANCIA:
El término instancia es un anglicismo de la palabra "instance", que significa "ejemplo". Cuando creamos un objeto, tomamos como modelo a una clase, por tanto, instancia y objeto son sinónimos.

VARIABLES DE INSTANCIA:
("Instance Variables"). De aquí en adelante las llamaremos IV. Son las variables que contienen un objeto creado.

VARIABLES DE INSTANCIA EXPORTABLES:
("Exported Instance Variables"). De aquí en adelante las llamaremos EIV. Son aquellas IV que podemos accesar.

METODOS Y MENSAJES:

Los métodos corresponden al código que opera sobre las IV, mientras que los mensajes corresponden a los mecanismos de comunicación entre los objetos.

8.1.1 Sintaxis para creación y manipulación de objetos

Para la creación y el manejo de objetos se debe aplicar la siguiente sintaxis:

SINTAXIS	DESCRIPCION
Objeto:=<Función de creación>	Creación de un objeto.
<Objeto>:<EIV>	Acceso a una EIV. El nombre de un objeto y su EIV forman un único nombre (cómo si fuese el nombre de una variable).
<Objeto>:<EIV>:=<Valor>	Modificación de una EIV. Usted especifica el nombre del objeto y de la EIV y seguidamente asigna un contenido a ésta.
<Objeto>:<Mensaje>[(<Parámetros>)]	Envío de métodos y mensajes.

Tabla 8.1 Sintaxis para creación y manipulación de objetos

8.2 LA CLASE TBCOLUMN

La clase TBCOLUMN sólo permite crear un objeto, el cual contiene información necesaria para tener control sobre una columna de una tabla. Un objeto creado por esta clase es solamente la definición de una columna y sus elementos que será visualizada por el objeto TBROWSE. La instancia que crea es TBCOLUMNNEW() y su sintaxis es:

TBCOLUMNNEW(<Título>,<Bloque>)

Donde <Título> es el título de la columna y <Bloque> es el CB que será evaluado para recuperar los datos del objeto TBCOLUMN.

En esta clase están disponibles las siguientes EIV, pudiendo todas tener valores asignados.

cargo
Puede contener un valor de cualquier tipo, no utilizado por el objeto. Esta EIV es un espacio disponible para que el programador añada una información al objeto y la recupere posteriormente.

block
Esta EIV debe contener el CB responsable de la recuperación de los datos de la columna. El CB no recibe ningún argumento cuando es evaluado.

colorblock
Contiene un CB que será responsable del color de los datos visualizados. Es evaluado cada vez que la EIV block sea enviada al objeto TBCOLUMN.

colsep
Contiene una cadena de caracteres que define el separador vertical de esta columna y que quedará a la izquierda de ella. Para aquellas columnas que no tengan especificado este valor, se usará el separador de columnas especificado por la EIV colsep del objeto TBROWSE.

footsep
Contiene una cadena de caracteres que será usada como separador de

pie de página de la columna.

headsep
Contiene una cadena de caracteres que será usada como separador de título de la columna.

defcolor
Esta EIV tiene una matriz con dos valores, los cuales hacen referencia a la tabla de colores de TBROWSE. El primero define el color relativo a los títulos de las columnas, pies de página y datos que no están siendo evaluados. El segundo define el color del dato de la celda actual, los valores por omisión son 1 y 2 respectivamente.

footing
Contiene una cadena de caracteres que será usada como pie de página de la columna.

heading
Contiene una cadena de caracteres que será usada como título para la columna.

width
Contiene una valor que determina el ancho de la columna. Si se omite, el ancho será determinado después de la evaluación de la EIV block.

8.3 LA CLASE TBROWSE

La clase TBROWSE es una constructora de mecanismos orientados al manejo de tablas. La extracción de datos y el posicionamiento en los mismos se pueden efectuar a través de CB, proporcionando elegancia y flexibilidad en el tipo de comportamiento que se desea efectuar.

8.3.1 Funciones de creación

La clase TBROWSE contiene dos funciones de creación: una para trabajar con archivos de datos (.DBF) y otra para la manipulación de datos diversos:

TBROWSEDB(<LinSup>,<ColSup>,<LinInf><ColInf>)

TBROWSENEW(<LinSup>,<ColSup>,<LinInf><ColInf>)

Los valores de entrada son de tipo numérico y corresponden a las coordenadas especificadas: (Línea Superior, Columna Superior, Línea Inferior, Columna Inferior).

La función TBROWSEDB() crea un objeto dentro de las coordenadas especificadas. Este objeto establece algunas variables para la utilización con los archivos .DBF. Esta es una diferencia entre ella y la función TBROWSENEW(). Al ser creado, este objeto no contiene ninguna columna objeto. Para cada campo del archivo .DBF que se desee mostrar, se debe añadir una columna al objeto.

La función TBROWSENEW() crea un nuevo objeto dentro de las coordenadas específicas. Este objeto no contiene ninguna columna ó CB, que deberán ser especificados por el usuario.

Dentro de la clase TBROWSE están disponibles las siguientes EIV. Aquellas que pueden recibir un valor se indican con un asterisco (*).

autolite(*)
Contiene un valor lógico que si es (.T.), indica al método de estabilización que resalte la celda actual.

cargo (*)
Puede contener un valor de cualquier tipo no utilizado por el objeto. Funciona como un espacio disponible para que el usuario añada alguna información al objeto y pueda recuperarla posteriormente.

colorespc(*)
Contiene una cadena de caracteres con la definición de colores a utilizar por el objeto TBROWSE. Si no se especifica la cadena, se asume el valor actual de la función SETCOLOR().

colcount
Contiene un número que indica la cantidad de columnas de TBROWSE. Cada columna de TBROWSE debe tener un objeto TBCOLUMN asociado a ella.

rowcount
Contiene un número que indica la cantidad de líneas (útiles) de TBROWSE, sin considerar las columnas de cabeceras y pies de página.

colpos(*)
Contiene un número indicando la columna sobre la cual está posicionado el cursor de TBROWSE.

rowpos(*)
Contiene un valor que indica la línea sobre la cual está situado el cursor de TBROWSE.

colsep(*)
Contiene una cadena de caracteres que define el separador de columnas para aquellas que no tienen un separador propio.

headsep(*)
Contiene una cadena de caracteres que define el separador de encabezamiento para las columnas que no tienen uno propio.

footsep(*)
Contiene una cadena de caracteres que define el separador del encabezamiento para las columnas que no tienen uno propio.

freeze(*)
Contiene un número que indica la cantidad de columnas a ser congeladas a partir de la primera columna de la izquierda y que permanecerán fijas en pantalla.

gobottomblock(*)
Contiene un CB que es ejecutado en respuesta al mensaje gobottom(). Tanto ésta como la EIV gotopblock deben usarse cuando un archivo .DBF se está visualizando.

gotopblock(*)
Contiene un CB que se ejecuta en respuesta al mensaje gotop(). El CB debe tener órdenes que sitúen al cursor en el primer registro del archivo.

hitbottom(*)
Contiene un valor lógico (.T.) que indica si se intentó pasar del último registro disponible.

nbottom(*)
Contiene un valor que indica la última línea usada para la visualización de TBROWSE.

leftvisible
Indica el número de la columna no congelada más a la izquierda de TBROWSE.

ntop(*)
Contiene un valor que indica la primera línea usada para la visualización de TBROWSE.

nleft(*)
Contiene un valor que indica la primera columna (de la pantalla) utilizada para la visualización de TBROWSE.

nright(*)
Contiene un valor que indica la última columna (de la pantalla) utilizada para la visualización de TBROWSE.

rightvisible
Indica el número de la columna no congelada más a la derecha de TBROWSE.

skipblock(*)
Contiene un CB que es responsable del reposicionamiento de datos. Normalmente el CB contiene una llamada a una función de usuario que ejecuta el comando SKIP. El CB es evaluado con el paso de un número positivo o negativo, que indica la cantidad de registros o líneas que se saltarán.

stabled
Tiene un valor lógico que, si es (.T.), indica que el objeto está estable. Este valor será (.F.) en caso contrario. Por acuerdo, TBROWSE se considera estable cuando todos los datos sean recuperados y exhibidos, el cursor sea reposicionado y la celda actual destacada. Cuando se envían mensajes de desplazamiento, esta EIV toma el valor (.F.) . La estabilización se realiza con el mensaje STABILIZE().

A continuación se describen los métodos disponibles para esta clase. Están clasificados en métodos de movimiento del cursor y métodos diversos.

8.3.2 Métodos del movimiento del cursor

up()
Desplaza el cursor un registro hacia arriba. Si el cursor estuviera en el primero, hittbottom sería puesto a (.T.). Si el cursor está en la primera línea de la ventana se producirá un desplazamiento vertical para permitir la visualización de la línea anterior.

down()
Desplaza el cursor una línea hacia abajo. Si el cursor estuviera en el último registro, hitbottom será (.T.). Si el cursor estuviera en la última línea de la ventana se produciría un desplazamiento vertical para permitir la visualización de la siguiente línea.

right()
Desplaza el cursor una columna a la derecha. Si el cursor estuviera en la última columna (de la pantalla), las columnas serían desplazadas hacia la izquierda, permitiendo la entrada de la columna inmediatamente posterior en la pantalla.

left()
Desplaza el cursor una columna a la izquierda, sin entrar en las columnas congeladas. Si el cursor ya estuviera en la primera columna (de la ventana), las columnas serán desplazadas hacia la derecha, permitiendo la entrada de la columna inmediata anterior en la pantalla.

home()
Sitúa el cursor en la primera columna no congelada (a la izquierda) visible en la ventana.

end()
Desplaza el cursor a la última columna (a la derecha) visible de la ventana.

panhome()
Desplaza el cursor a la primera columna de TBROWSE, causando un movimiento lateral de la columnas para visualizar la primera.

panend()
Desplaza el cursor a la última columna de TBROWSE, causando un movimiento lateral de la columnas para visualizar la última.

panright()
Causa un desplazamiento lateral de la ventana para mostrar las columnas que estaban ocultas a la derecha, procurando mantener el cursor en su posición actual.

panleft()
Causa un desplazamiento lateral de la ventana para mostrar las columnas que estaban ocultas a la izquierda, procurando mantener el cursor en su posición actual.

gotop()
Desplaza el cursor hacia el inicio del archivo lógico mediante la evaluación del CB especificado por gotopblock.

gobottom
Desplaza el cursor hasta el fin del archivo lógico mediante la evaluación del CB especificado por gotopblock.

pagedown()
Si el último registro de la tabla estuviera visible en la pantalla, sitúa el cursor sobre él. Si no está visible, retrocede el cursor una pantalla.

8.3.3 Métodos diversos

addcolumn(<tbcol>)
Este método añade un nuevo objeto TBCOLUMN al objeto TBROWSE e incrementa en uno el valor de COLCOUNT.

colorrect(<matriz>,<colores>)
Altera el color de un grupo rectangular de celdas cuyas coordenadas están especificadas por la matriz de cuatro números y que se refieren a la posición de las celdas en la pantalla. <colores> es una matriz con dos números que indican los colores que serán usados como color normal y resaltado del rectángulo.

colwidth(<nPOS>)
Devuelve el ancho de la columna de número especificado.

configure()
Este método forza la reevaluación de todas las IV, reconfigurando los valores internos de un objeto TBCOLUMN.

dehilite()
Desactiva la iluminación de la celda actual, debe usarse cuando autolite tenga valor falso.

hilite()
Resalta la celda actual, y debe usarse cuando autolite tenga valor falso.

delcolumn(<nCol>)
Elimina una columna de número especificado.

getcolumn(<col>)
Devuelve el objeto TBCOLUMN especificado por <col>.

inscolumn(<pos>,<col>)
Inserta una columna en la posición especificada.

setcolumn(<col1>,<col2>)
Este método sustituye el objeto TBCOLUMN especificado por <col1>, por el objeto TBCOLUMN especificado por <col2>, y devuelve el valor del objeto TBCOLUMN actual.

invalidate()
Hace que la próxima estabilización rediseñe todo el TBROWSE.

refreshall()
Causa una revisualización de todas las líneas y columnas durante el próximo proceso de estabilización de TBROWSE.

stabilize()
Ejecuta el proceso de estabilización de TBROWSE. La estabilización es incremental , es decir, cada vez que se envía un mensaje de estabilización, se ejecuta una parte del proceso de estabilización. De esta forma, el proceso puede ser interrumpido y valorado.

8.3.4 Creación y ejecución de un Browse

A diferencia de DBEDIT(), TBROWSE es totalmente pasivo. Así, si usted quiere que al pulsar la tecla FLECHA DERECHA se desplace el cursor una columna a la derecha, tendrá que especificárselo a TBROWSE mediante el envío de un método.

Los pasos necesarios para crear la visualización de una tabla completa son:

1. Crear un objeto Tbrowse
2. Crear instancias de la Clase TBCOLUMN (tantas como columnas)
   deseemos desplegar).
3. Enviar mensajes al objeto TBROWSE.
4. Enviar mensajes para desplegar información y cualquier tipo de mensaje soportado por las instancias creadas.

Los pasos anteriores se ejemplifican en el programa prueba81.prg. Obsérvese que la comprobación de la tecla pulsada está fuera del cuerpo del programa para darle mayor claridad.

Para este programa y los posteriores se usará el archivo de base de datos "archi.dbf" que tiene la siguiente estructura:

NOMBRE	TIPO	ANCHO	DESCRIPCION
CVE	char	6	Clave del empleado
NOMB	char	35	Nombre
SLARIO	num	9.2	Salario
PLANT	log	1	Trabajador de planta .T./.F.

```
/* Programa:          PRUEBA81.PRG    Hecho en Clipper 5.01
   Creación:          03/01/94        Por:  Jorge Castaño
   Ultima revisión:   06/01/94        Por:  Fernando Tecuapacho
   Propósito:  Ejemplificar el uso de la Clase TBROWSE
               Para su ejecución este programa requiere del archivo: archi.dbf que
               tiene los siguientes campos: CVE, NOMB, SLARIO y PLANT      */
```

```
#INCLUDE "INKEY.CH"
CLS
USE archi ALIAS ar NEW
@9,7 TO 22,74 DOUBLE
oTAB:=TBROWSEDB(10,8,22,73)   //  Crea un objeto llamado oTAB

/*   Las siguientes lineas crean los separadores  */
oTAB:HEADSEP:=CHR(205)+CHR(209)+CHR(205)
oTAB:COLSEP:= CHR(32) +CHR(179)+CHR(32)
oTAB:FOOTSEP:=CHR(205)+CHR(207)+CHR(205)

/*  Se crean las instancias de la Clase TBCOLUMN y se añaden mensajes al objeto oTAB  */
COLUMNA:=TBCOLUMNNEW("CLAVE",{||CVE})
oTAB:ADDCOLUMN(COLUMNA)
COLUMNA:=TBCOLUMNNEW("NOMBRE",{||NOMB})
oTAB:ADDCOLUMN(COLUMNA)
COLUMNA:=TBCOLUMNNEW("SALARIO",{||SLARIO})
oTAB:ADDCOLUMN (COLUMNA)
COLUMNA:=TBCOLUMNNEW("PLANTA",{||PLANT})
oTAB:ADDCOLUMN (COLUMNA)

/*  Bucle principal.    Dentro de este bucle va cualquier tipo de mensaje
     soportado por las instancias creadas.  */

WHILE .T.
  WHILE NEXTKEY() ==0 .AND. .NOT. oTAB:STABILIZE()
  ENDD       // Se permanece en este ciclo hasta que se estabiliza el objeto
  Tecla:=INKEY(0)    /*     Se captura la tecla pulsada      */
  IF Tecla==K_ESC   //  Escape abandona el ciclo
    EXIT
  ELSE
    Pruebatecla(Tecla)
  ENDIF
ENDD
CLOSE ar
CLS
/*      Fin del cuerpo del programa.    */
```

```
FUNCTION Pruebatecla(Tecla)
/*  Las constantes usadas abajo están definidas en el archivo INKEY.CH  */
DO CASE
    CASE Tecla ==K_UP;          oTAB:UP()
    CASE Tecla ==K_DOWN         ;oTAB:DOWN()
    CASE Tecla ==K_LEFT         ;oTAB:LEFT()
    CASE Tecla ==K_RIGHT        ;oTAB:RIGHT()
    CASE Tecla ==K_PGUP         ;oTAB:PAGEUP()
    CASE Tecla ==K_PGDN         ;oTAB:PAGEDOWN()
    CASE Tecla ==K_HOME         ;oTAB:HOME()
    CASE Tecla ==K_END          ;oTAB:END()
    CASE Tecla ==K_CTRL_PGUP ;oTAB:GOTOP()
    CASE Tecla ==K_CTRL_PGDN ;oTAB:GOBOTTOM()
    OTHERWISE
    TONE(120,4)       //  Cualquier tecla que no haya sido asociada a un
                      //  método causará la emisión de un pitido.
ENDCASE
RETURN(NIL)
```

Al ejecutar el programa se verá una pantalla como la mostrada en la figura 8.1

CLAVE	NOMBRE	SALARIO	PLANTA
B09500	RAMON JUAREZ	1760.00	T
B10479	MARIA ESCOBAR	1540.44	F
B10897	ISABEL GARCIA	1485.00	F
B11000	TERESA MORALES	1650.00	T
B11100	LOURDES MONTOYA	1650.00	T
B11478	ADRIAN CAMACHO	1540.44	T
C09030	RODOLFO MARROQUIN	2860.00	T
C10900	IRMA BARAJAS	2750.22	T
C12508	FERNANDO TECUAPACHO	2420.00	T
C12579	JORGE CASTAÑO	2420.00	T

Figura 8.1 Aspecto del TBROWSE en pantalla

8.4 TECNICAS DE MANEJO PARA LA CLASE TBROWSE

Basándonos en el programa prueba81.prg, se explicarán algunas de las técnicas utilizadas en el manejo del TBROWSE.

8.4.1 Creación del objeto TBROWSE sin importar la estructura del archivo .DBF

En caso de que se quiera visualizar todos los campos de un archivo .DBF, se puede reducir el código del programa prueba81.prg y se obtiene el programa prueba82. Las columnas se agregan al objeto Tbcolumn mediante un ciclo For...Next. La creación del objeto oTAB y la adición de las columnas se incluyen dentro de la función Creabrowse().

```
/* Programa:          PRUEBA82.PRG          Hecho en Clipper 5.01
      Creación:       03/01/94              Por:  Jorge Castaño
      Ultima revisión: 06/01/94             Por:  Fernando Tecuapacho

      Propósito:  Ejecutar un browse independientemente de la estructura del
                  archivo .DBF
                  Este programa requiere de un archivo con nombre: archi.dbf   */

#INCLUDE "INKEY.CH"
CLS
USE archi ALIAS ar NEW
//  Crea un objeto llamado oTAB mediante la función de usuario Creabrowse
oTAB:=Creabrowse(10,10,MAXROW()-1,MAXCOL()-1)
/*      Bucle principal.    */
WHILE .T.
  WHILE NEXTKEY() ==0 .AND. .NOT. oTAB:STABILIZE()
  ENDD      // Se permanece en este ciclo hasta que se estabiliza el objeto
  Tecla:=INKEY(0)   /*     Se captura la tecla pulsada     */
  IF Tecla==K_ESC
    EXIT        // Escape abandona el ciclo
  ELSE
    Pruebatecla(Tecla)
  ENDIF
ENDD
CLOSE ar
```

```
CLS
//  Fin del cuerpo del programa.  A continuación van las funciones de usuario

FUNCTION Creabrowse(Ri,Ci,Rf,Cf)
@Ri-1,Ci-1 TO Rf,Cf+1 DOUBLE
oTAB :=TBROWSEDB(Ri,Ci,Rf,Cf)   //  Crea el objeto oTAB
/*   Las siguientes líneas crean los separadores  */
oTAB:HEADSEP:=CHR(205)+CHR(209)+CHR(205)
oTAB:COLSEP:= CHR(32) +CHR(179)+CHR(32)
oTAB:FOOTSEP:=CHR(205)+CHR(207)+CHR(205)
/*      Se crean las instancias de la Clase TBCOLUMN y se añaden mensajes
        al objeto oTAB          */
FOR I =1 TO FCOUNT()
  oTAB:ADDCOLUMN(TBCOLUMNNEW(FIELDNAME(I),;
  FIELDBLOCK(FIELDNAME(I),SELE())))
NEXT
RETURN(oTAB)

FUNCTION Pruebatecla(Tecla)
/*  Las constantes usadas abajo están definidas en el archivo INKEY.CH  */
DO CASE
    CASE Tecla ==K_UP            ;oTAB:UP()
    CASE Tecla ==K_DOWN          ;oTAB:DOWN()
    CASE Tecla ==K_LEFT          ;oTAB:LEFT()
    CASE Tecla ==K_RIGHT         ;oTAB:RIGHT()
    CASE Tecla ==K_PGUP          ;oTAB:PAGEUP()
    CASE Tecla ==K_PGDN          ;oTAB:PAGEDOWN()
    CASE Tecla ==K_HOME          ;oTAB:HOME()
    CASE Tecla ==K_END           ;oTAB:END()
    CASE Tecla ==K_CTRL_PGUP ;oTAB:GOTOP()
    CASE Tecla ==K_CTRL_PGDN ;oTAB:GOBOTTOM()
    OTHERWISE
    TONE(120,4)     //Cualquier tecla que no haya sido asociada a un
                    // método causará la emisión de un pitido.
ENDCASE
RETURN(NIL)
```

8.4.2 Desplazamiento de columnas

Cuando se está consultando un archivo con muchas columnas, es sumamente útil el poder cambiar una columna de posición. En Clipper 5.01 esto es una tarea muy simple cuya técnica se describe a continuación.

Se define una variable para controlar el movimiento de las columnas "Colmueve" y se crea una función "Muevecol()" que será accionada tecleando <ALT><M>. La función comprueba el valor de la variable; si es 0, crea una variable para almacenar las características de la columna actual y a continuación la elimina, estableciendo el valor de la variable a 1.

Desplace el cursor a la columna donde quiera insertar la columna eliminada y pulse nuevamente <ALT><M>. La función comprobará que el valor de la variable sea igual a 1 e insertará la columna en esa posición.

Abajo se presenta el listado del programa. En la figura 8.2 se muestra la columna CVE desplazada después de la columna NOMB.

```
/* Programa:        PRUEBA83.PRG     Hecho en Clipper 5.01
   Creación:        03/01/94         Por:   Jorge Castaño
   Ultima revisión: 06/01/94         Por:   Fernando Tecuapacho

   Propósito: Ejemplificar la técnica de desplazamiento de columnas.
              Este programa requiere de un archivo con nombre: archi.dbf
              (Sin importar su estructura)                              */

#INCLUDE "INKEY.CH"
PUBLIC Nuevacol,Posicion,Colmueve,Colfuera
Colmueve:=Colfuera:=0

CLS
USE archi ALIAS ar NEW
// Crea un objeto llamado oTAB mediante la función de usuario Creabrowse
oTAB:=Creabrowse(10,10,MAXROW()-1,MAXCOL()-1)

/*     Bucle principal.   */
WHILE .T.
  WHILE NEXTKEY() ==0 .AND. .NOT. oTAB:STABILIZE()
```

```
  ENDD                // Se permanece en este ciclo hasta que se estabiliza el objeto
  Tecla:=INKEY(0)   /* Se captura la tecla pulsada      */
  IF Tecla==K_ESC
    EXIT              // Escape abandona el ciclo
  ELSE
    Pruebatecla(Tecla)
  ENDIF
ENDD
CLOSE ar
CLS
RETURN       //Fin del cuerpo del programa.  A continuación van las funciones de usuario

FUNCTION Creabrowse(Ri,Ci,Rf,Cf)
@Ri-1,Ci-1 TO Rf,Cf+1 DOUBLE
oTAB :=TBROWSEDB(Ri,Ci,Rf,Cf)   //  Crea el objeto oTAB
/*    Las siguientes lineas crean los separadores  */
oTAB:HEADSEP:=CHR(205)+CHR(209)+CHR(205)
oTAB:COLSEP:= CHR(32) +CHR(179)+CHR(32)
oTAB:FOOTSEP:=CHR(205)+CHR(207)+CHR(205)
/*       Se crean las instancias de la Clase TBCOLUMN y se añaden al objeto oTAB        */
FOR I=1 TO FCOUNT()
  oTAB:ADDCOLUMN(TBCOLUMNNEW(FIELDNAME(I),;
  FIELDBLOCK(FIELDNAME(I),SELE())))
NEXT
RETURN(oTAB)

FUNCTION Pruebatecla(Tecla)
/*  Las constantes usadas abajo están definidas en el archivo INKEY.CH   */
DO CASE
    CASE Tecla ==K_UP                    ;oTAB:UP()
    CASE Tecla ==K_DOWN                  ;oTAB:DOWN()
    CASE Tecla ==K_LEFT                  ;oTAB:LEFT()
    CASE Tecla ==K_RIGHT                 ;oTAB:RIGHT()
    CASE Tecla ==K_PGUP                  ;oTAB:PAGEUP()
    CASE Tecla ==K_PGDN          ;oTAB:PAGEDOWN()
    CASE Tecla ==K_HOME                  ;oTAB:HOME()
    CASE Tecla ==K_END                   ;oTAB:END()
    CASE Tecla ==K_CTRL_PGUP ;oTAB:GOTOP()
    CASE Tecla ==K_CTRL_PGDN ;oTAB:GOBOTTOM()
    CASE Tecla ==K_ALT_M                 ;Muevecol(oTAB)
    OTHERWISE
    TONE(120,4)              //Cualquier tecla que no haya sido asociada a un
```

```
                                    // método causará la emisión de un pitido.
ENDCASE
RETURN(NIL)

FUNCTION Muevecol(oTAB)
//  Mediante esta función se puede borrar una columna e insertarla en otra posición
IF Colmueve ==0    //   Se borrará la columna
  Nuevacol =oTAB:GETCOLUMN(oTAB:COLPOS)
  oTAB:DELCOLUMN(oTAB:COLPOS)
  Colmueve =1
ELSE               //   Se insertará la columna
  Posicion =oTAB:COLPOS
  oTAB:INSCOLUMN(Posicion,Nuevacol)
  Colmueve =0
ENDIF
RETURN(NIL)
```

NOMB	CVE	SLARIO	PLANT
RAMON JUAREZ	B09500	1760.00	T
MARIA ESCOBAR	B10479	1540.44	F
ISABEL GARCIA	B10897	1485.00	F
TERESA MORALES	B11000	1650.00	T
LOURDES MONTOYA	B11100	1650.00	T
ADRIAN CAMACHO	B11478	1540.44	T
RODOLFO MARROQUIN	C09030	2860.00	T
IRMA BARAJAS	C10900	2750.22	T
FERNANDO TECUAPACHO	C12508	2420.00	T
JORGE CASTAÑO	C12579	2420.00	T
HUMBERTO MAYEN	D01389	4950.00	T

Figura 8.2 La columna CVE desplazada después de NOMB

8.4.3 Filtrado de registros

Un punto muy importante relacionado con TBROWSE es como mostrar solo registros que cumplan con una condición de búsqueda.

La técnica consiste en controlar el valor de tres IV del objeto TBROWSE:

GOTOPBLOCK	Indica el primer registro a visualizar
GOBOTTOMBLOCK	Indica el último registro a visualizar
SKIPBLOCK	Controla el avance de los registros

La finalidad del siguiente programa es la de mostrar solo los empleados de planta. El código está basado en el del programa prueba82.prg

```
/* Programa:        PRUEBA84.PRG     Hecho en Clipper 5.01
   Creación:        11/01/94         Por:  Jorge Castaño
   Ultima revisión: 17/01/94         Por:  Fernando Tecuapacho

   Propósito:       Ejecutar un browse con un filtrado de registros.
                    Este programa requiere de un archivo con nombre:
                    ARCHI.DBF que tenga un campo llamado PLANT      */

#INCLUDE "INKEY.CH"
IF (FILE("archipla.ntx"))
  USE archi ALIAS ar INDEX archipla NEW
ELSE
  USE archi ALIAS ar NEW
  INDEX ON PLANT TO archipla
ENDIF
WHILE .T.
  CLS
  Tipo :=" "
  @8,10 SAY "" Empleados de planta ? S/N  (Para salir <Esc>) " GET Tipo PICT "!"
VALID(Tipo$"SN")
  READ
  IF LASTKEY() =K_ESC
    EXIT          // Con Esc abandona el ciclo
  ELSEIF Tipo ="S"
    Tipo :=.T.
  ELSE
    Tipo :=.F.
  ENDIF
  SEEK Tipo
  //   Crea un objeto llamado oTAB mediante la función de usuario Creabrowse
  //   filtrando aquellos registros que cumplan con la condición
```

```
  oTAB:=Creabrowse(10,10,MAXROW()-1,MAXCOL()-1,Tipo)
  /*       Bucle principal.      */
  WHILE .T.
        WHILE NEXTKEY() ==0 .AND. .NOT. oTAB:STABILIZE()
        ENDD        // Se permanece en este ciclo hasta que se estabiliza el objeto
        Tecla:=INKEY(0)    /*     Se captura la tecla pulsada      */
        IF Tecla==K_ESC
          EXIT          // Escape abandona el ciclo
        ELSE
          Pruebatecla(Tecla)
        ENDIF
  ENDD
ENDD
CLOSE ar
CLS
RETURN       //Fin del cuerpo del programa.  A continuación van las funciones de usuario

FUNCTION Creabrowse(Ri,Ci,Rf,Cf,Tipo)
@Ri-1,Ci-1 TO Rf,Cf+1 DOUBLE
oTAB :=TBROWSEDB(Ri,Ci,Rf,Cf)    //  Crea el objeto oTAB
/*    Las siguientes lineas crean los separadores   */
oTAB:HEADSEP:=CHR(205)+CHR(209)+CHR(205)
oTAB:COLSEP:= CHR(32) +CHR(179)+CHR(32)
oTAB:FOOTSEP:=CHR(205)+CHR(207)+CHR(205)
/*     Se crean las instancias de la Clase TBCOLUMN y se añaden mensajes
         al objeto oTAB          */
FOR I =1 TO FCOUNT()
   oTAB:ADDCOLUMN(TBCOLUMNNEW(FIELDNAME(I),;
   FIELDBLOCK(FIELDNAME(I),SELE())))
NEXT
oTAB:GOTOPBLOCK={|| Inirango(Tipo) }    //  El inicio
oTAB:GOBOTTOMBLOCK:={|| Finrango(Tipo) }    //  El fin
oTAB:SKIPBLOCK:={|Paramover| Muevereg(Paramover,Tipo)}
RETURN(oTAB)

FUNCTION Inirango(Tipo)
/*  Obtiene el primer registro que cumpla con la condición   */
RETURN({|| DBSEEK(Tipo)})

FUNCTION Finrango(Tipo)
/*  Obtiene el último registro que cumpla con la condición   */
RETURN({|| DBSEEK(Ultimllave(Tipo),.T.),DBSKIP(-1)})
```

```
FUNCTION Ultimllave(Tipo)          // Esta función obtiene la siguiente
LOCAL cTipo :=VALTYPE(Tipo)        // llave consecutiva al límite
LOCAL Siguiente
DO CASE
  CASE(cTipo =="C")
       Tipo :=STUFF(Tipo,LEN(Tipo),1,CHR(ASC(RIGHT(Tipo,1)) +1))
  CASE(cTipo =="N".OR. cTipo =="D")
       Tipo++
  CASE(cTipo =="L")
       Tipo := !Tipo
ENDCASE
Siguiente :=Tipo
RETURN(Siguiente)

FUNCTION Muevereg(Paramover,Tipo)
LOCAL Regmovid
Regmovid :=0
IF Paramover ==0 .OR. LASTREC() ==0
  DBSKIP(0)
ELSEIF Paramover >0 .AND. RECNO() !=LASTREC() +1
  WHILE Regmovid <=Paramover .AND. ! EOF() .AND. &(INDEXKEY(0)) ==Tipo
    DBSKIP(1)
    Regmovid++
  ENDD
  DBSKIP(-1)           // Siempre devuelve un registro, pues el bucle While
  Regmovid--           // termina cuando estamos un registro adelante del rango
ELSEIF Paramover <0
  WHILE Regmovid >=Paramover .AND. ! BOF() .AND. &(INDEXKEY(0)) ==Tipo
    DBSKIP(-1)
    Regmovid--
  ENDD
  IF ! BOF()           // Siempre devuelve un registro, pues el bucle While
    DBSKIP(1)          // termina cuando estamos un registro arriba del rango,
  ENDIF                // en el caso de que no sea inicio de archivo
    Regmovid++
  ENDIF
RETURN(Regmovid)

FUNCTION Pruebatecla(Tecla)
/*  Las constantes usadas abajo están definidas en el archivo INKEY.CH  */
DO CASE
    CASE Tecla ==K_UP              ;oTAB:UP()
```

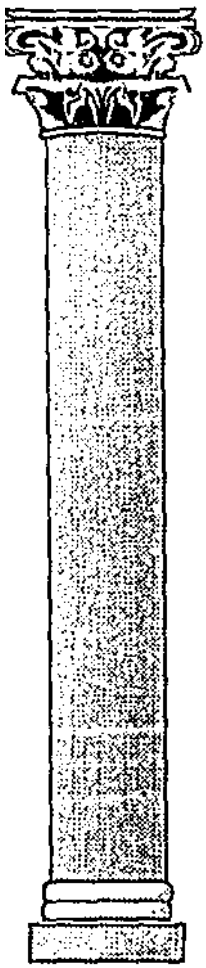
```
    CASE Tecla ==K_DOWN          ;oTAB:DOWN()
    CASE Tecla ==K_LEFT          ;oTAB:LEFT()
    CASE Tecla ==K_RIGHT         ;oTAB:RIGHT()
    CASE Tecla ==K_PGUP          ;oTAB:PAGEUP()
    CASE Tecla ==K_PGDN          ;oTAB:PAGEDOWN()
    CASE Tecla ==K_HOME          ;oTAB:HOME()
    CASE Tecla ==K_END           ;oTAB:END()
    CASE Tecla ==K_CTRL_PGUP     ;oTAB:GOTOP()
    CASE Tecla ==K_CTRL_PGDN     ;oTAB:GOBOTTOM()
    OTHERWISE
    TONE(120,4)      // Cualquier tecla que no haya sido asociada a un
                     // método causará la emisión de un pitido.
ENDCASE
RETURN(NIL)
```

Obsérvese que el cuerpo del programa fue creado para un campo llave de tipo lógico, pero las funciones de usuario sirven para cualquier tipo de campo. En la figura 8.3 se muestra el filtrado de los empleados que no son de planta.

CVE	NOMB	SLARIO	PLANT
B10479	MARIA ESCOBAR	1540.00	F
B10897	ISABEL GARCIA	1485.00	F
D04000	RAUL MARTINEZ	2860.00	F
E08002	RODOLFO SORIA	2640.00	F

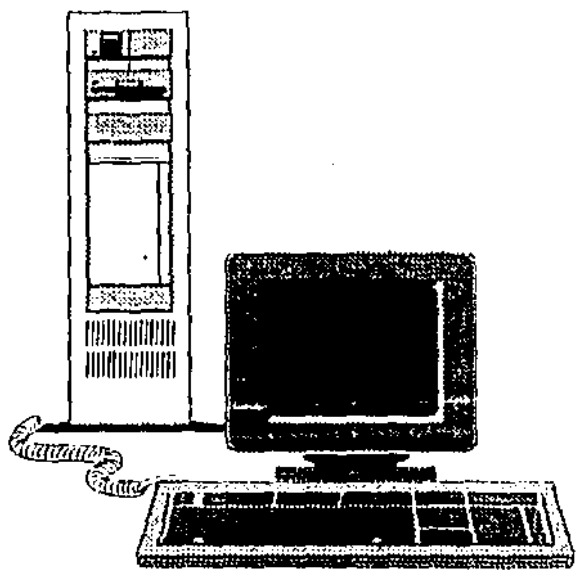
Figura 8.3 Aspecto del filtrado de los empleados que no son de planta

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO



CAPITULO 9

- PROGRAMACION EN AMBIENTE DE RED



ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES
A R A G O N

9.1 MANEJO DE ARCHIVOS EN RED DE AREA LOCAL

La función principal de una red de micros es la de compartir los recursos con otros usuario sin necesidad de que cada quien tenga su propio equipo.

Al hablar de compartir recursos nos referimos a que varios usuarios puedan utilizar, por ejemplo, la misma impresora, sin necesidad de estar desconectándola de donde esté para traerla y conectarla a su propia computadora. También podemos compartir un mismo disco en donde todos los usuarios estén accediendo diferentes directorios y archivos.

Pero aquí se nos ocurre preguntar ¿por qué no compartir los mismos archivos?.

Precisamente esta pregunta es la que nos involucra con el concepto de multiusuario. Claro, es posible compartir los mismos archivos por varios usuarios pero con ciertas restricciones físicas y lógicas.

Una restricción física podría ser la imposibilidad de acceder la misma área de disco por dos usuarios simultáneamente.

Una restricción lógica puede estar basada en la integridad de los datos. Así dos usuarios no utilizarán simultáneamente el mismo registro para efectuar una modificación sobre los datos.

El objetivo de esté capitulo es explicar al usuario como se comporta un programa multiusuario y su estructura al momento de ser aplicado.

Estrictamente hablando, la palabra multiusuario indica que el programa puede ser utilizado por muchos usuarios, pero no indica si al mismo tiempo, en la misma computadora, etc.

Básicamente el concepto interesante de un sistema multiusuario es habilitar a varias personas en diferentes máquinas para consultar y/o alterar los datos de una cierta base de datos, tratando de evitar problemas potenciales en cuanto a la utilización de la base de datos.

Un problema potencial puede darse si un usuario está modificando en cierto momento el precio de un producto y algún otro intenta consultar dicho precio, es posible que este último obtenga el precio anterior a la modificación , con lo cual se generaría un "error de información".

Para resolver este problema se debe se crear un mecanismo y garantizar que la información consultada en cualquier momento sea la actualizada.

Este mecanismo es precisamente la capacidad de algunos lenguajes de programación para identificar cuando algún usuario está utilizando algún registro y poder tomar un criterio de ejecución; como puede ser el esperarse a que se desocupe el registro o cancelar la operación.

El compilador de Clipper nos permite hacer programación en ambiente multiusuario y esto no es nada nuevo. El capítulo 10 del manual de Summer 87, explica brevemente los cambios para lograr este tipo de programación.

El compilador de Clipper tiene dentro de sus funciones la de permitir crear programas multiusuarios controlando los accesos a los registros de datos.

Básicamente con estas funciones podemos garantizar que no existan problemas potenciales como los indicados anteriormente.

Clipper es compatible con el DOS 3.1 o superior para multiusuario. Las consultas de Clipper a la red son a través del DOS, por eso funciona en redes con sistemas operativos basados en el DOS estándar.

9.2 TECNICA Y FILOSOFIA

Desarrollar aplicaciones que trabajen en una Red Local puede ser un proyecto sencillo o catastrófico. Es importante considerar varios elementos de una aplicación en Red, no solo es cuestión de seguir los pasos que se describen en el manual de Clipper, debido a que efectivamente una aplicación puede operar en Red si establecemos el uso de apertura de archivos en forma compartida, bloquear un archivo un registro, etc.

Los aspectos que deben considerarse para el desarrollo de aplicaciones en ambientes multiusuarios son los siguientes:

- Maximizar Concurrencias
- Asegurar Integridad de Información
- Optimizar Efectividad

Maximizar Concurrencias:

El manejo de la concurrencia es un elemento importante para el usuario de nuestras aplicaciones. El primer punto a considerar para el manejo de concurrencia es evitar el estancamiento (Deadlock). Por ejemplo, imagine que la terminal A abre el archivo X y posteriormente intenta abrir el archivo Y en forma exclusiva; mientras tanto, la terminal B ha abierto el archivo Y y posteriormente intenta abrir el archivo X en forma exclusiva. Como la terminal A tiene abierto al archivo X, la terminal B no puede abrirlo en forma exclusiva; y como, la terminal B tiene abierto al archivo Y, la terminal A no puede abrirlo en forma exclusiva. Ambas estaciones fallarán en su operación.

Como regla No. 1, nunca trate de abrir los archivos en forma exclusiva. Desarrolle funciones que le permitan abrir los archivos dependiendo de factores como número de intentos. Hasta estar seguro de que puede tener acceso a un archivo, active los índices.

Asegurar Integridad de información:

La información es elemento más valioso de una aplicación, de tal forma que la integridad de la misma es un punto que debemos tratar con mucha delicadeza.

La regla No. 2, no efectúe GET's sobre campos directamente, es necesario conservar los datos de acuerdo a la normalización de los archivos. Desarrolle funciones que permitan asegurar que el registro al que se va a modificar los datos, quede protegido de otra terminal y se remplacen valores incongruentes.

Evite el SET RELATION TO..., que apesar de ser un comando muy atractivo, puede llegar a ser muy destructivo en cuanto a la integridad de la información.

Optimizar Efectividad:

Los procesos en un servidor pueden ser pesados, de tal forma que es de suma importancia desarrollar con el objetivo de lograr la mayor efectividad posible en la ejecución de la aplicación.

Para lograr mayor efectividad considere las siguientes sugerencias:

- Limite la frecuencia de bloqueos.
- Reduzca la longitud de las expresiones de sus índices.
- Elimine o reduzca los campos memo.
- Cuando necesite procesar información en donde no tenga importancia el manejo de los índices ciérrelos.
- Evite a toda costa los filtros.
- Jamás empaque un archivo .Dbf, utilizar el concepto de reusabilidad.

9.3 ACCESO COMPARTIDO Y EXCLUSIVO

Clipper permite abrir un archivo para usarse en modo compartido o exclusivo. Un archivo que se abre para uso exclusivo, no puede dar acceso a otros usuarios. Por default Clipper abre archivos .dbf para uso exclusivo (los archivos índices se les asigna el mismo modo que el archivo de la base de datos asociado). Aun cuando cada tipo de archivo que usa Clipper se puede abrir en modo compartido o exclusivo, lo más importante cuando se diseña una aplicación de base de datos es garantizar que las bases de datos se abrirán y se podrán actualizar sin problemas se seguridad, ya que la pérdida o daño de las bases de datos es un problema más serio, que requiere más tiempo, esfuerzo y resultan mas costosas las reparaciones.

9.4 COMANDOS Y FUNCIONES PARA AMBIENTE MULTIUSUARIO

Clipper tiene comandos y funciones asociados al manejo de la red y se deben utilizar conjuntamente para crear funciones de usuario que nos permitan trabajar bases de datos en ambiente multiusuario.

El comando "SET EXCLUSIVE ON/OFF" nos da la pauta del ambiente que será utilizado:

SET EXCLUSIVE ON => ambiente monousuario, por default.
SET EXCLUSIVE OFF => ambiente multiusuario.

Aunque este mandato es soportado por la versión 5.01 de Clipper, se recomienda sustituirlo por las cláusulas EXCLUSIVE y SHARED del mandato USE

```
USE [<arch> [INDEX <Antx1> {,<Antx2>}]
    [EXCLUSIVE/SHARED]
    [ALIAS <alias>]
    [NEW]
    [READONLY]]
```

INDEX <Antx1>	{,<Antx2>}]Es la lista de archivos índices asociados al archivo <arch>. Recuerde que el máximo es de 15 archivos índice.
EXCLUSIVE/SHARED	Se emplea en redes y su finalidad es la de abrir ficheros para uso exclusivo de un usuario (EXCLUSIVE) o compartido por varios (SHARED).
ALIAS <alias>	Determina el nombre del alias del archivo.
NEW	Abre el archivo en la siguiente área de trabajo a la que se encuentra actualmente activa.
READONLY	Permite abrir el fichero con el atributo de solo lectura.

SET PRINTER
Este comando sirve para determinar la impresora: la local o la de red con la que trabajaremos. También sirve para desviar todas las salidas SAY, TEXT, etc. a un archivo de texto.

SET PRINTER TO <destino>=>	Impresora de red. Recuerde que en la instalación de algunos sistemas operativos de red se asignan nombres lógicos a las impresoras.
SET PRINTER TO Lpt1 =>	Impresora local.
SET PRINTER TO [Atxt]=>	Desviar salida a un archivo de texto.

UNLOCK
Libera el bloqueo de la base de datos activa o del registro en uso. Los bloqueos solamente podrán ser suprimidos por el mismo usuario de la red que los bloqueo.
UNLOCK [ALL]

Libera todos los bloqueos en todas las áreas de trabajo. Sin argumentos liberará el bloqueo de la base de datos que está en el área de trabajo actualmente seleccionada.

Las bases de datos abiertas en uso exclusivo no podrán ser desbloqueadas con UNLOCK.

Las funciones implicitas de Clipper son:

FLOCK()
Devuelve un valor lógico verdadero (.T.) si el bloque del archivo ha tenido éxito, para que cualquier otro acceso al mismo por parte de otro usuario quede denegado, a menos que sea para consultar. El archivo quedará bloqueado hasta que lo libere el usuario que lo bloqueó, usando el mandato UNLOCK o cerrando dicho archivo.

Algunos de los mandatos, que para operar en red, necesitan que el archivo este bloqueado se muestran en tabla 9.1.

Comando	Requerimiento
@...SAY...GET	RLOCK().
APPEND FROM	USE...EXCLUSIVE o FLOCK().
DELETE (uno)	RLOCK().
DELETE (varios)	USE...EXCLUSIVE o FLOCK().
PACK	USE...EXCLUSIVE.
RECALL (uno)	RLOCK().
RECALL (varios)	USE...EXCLUSIVE o FLOCK().
REINDEX	USE...EXCLUSIVE.
REPLACE (uno)	RLOCK().
REPLACE (varios)	USE...EXCLUSIVE o FLOCK().
UPDATE ON	USE...EXCLUSIVE o FLOCK().
ZAP	USE...EXCLUSIVE.

TABLA 9.1

RLOCK()
Bloquea o intenta bloquear un registro, permitiendo acceso al usuario que estableció el bloqueo a operaciones de lectura y escritura, mientras que los demás usuarios solo podrán tener acceso de lectura al registro.

RLOCK()
Devuelve verdadero (.T.) cuando han tenido éxito el bloqueo de un registro. Este registro permanecerá en este estado hasta que se bloque otro registro, o se haga UNLOCK, FLOCK o se cierre la base de datos.

NETERR()
Determina si un mandato o función de red ha tenido éxito al ejecutarse. Algunos de estos mandatos o funciones pueden ser USE...EXCLUSIVE, APPEND BLANK, FLOCK() o RLOCK().

NETERR()
Da valor verdadero (.T.) cuando han fallado alguno de los mandatos anteriores (Tabla 9.2).

Comando	Razón de la falla
USE...	USE ...EXCLUSIVE hecho por otro usuario (proceso).
USE...EXCLUSIVE	USE o USE...EXCLUSIVE hecho por otro usuario.
APPEND BLANK	FLOCK() por otro usuario o dos intentos de APPEND BLANK al mismo tiempo.

TABLA 9.2

Para utilizar las funciones para manejo compartido de archivos de una manera más robusta se dan algunos ejemplos implementando varias funciones, basándose en las funciones primitivas RLOCK() y FLOCK(). Dichas funciones básicamente se utilizan para lograr que el bloqueo de archivos o de registros se intente una serie de veces dependiendo del valor enviado como parámetro.

En seguida mostramos el código y una pequeña explicación de las funciones de usuario mencionadas anteriormente:

Net_use(file,ex_use,intento): la cual se encarga de intentar abrir la base de datos enviada como parámetro "file" con el modo definido en la variable "ex_use" y con el número de veces controlado por la variable "intento".

Utilizando la función de usuario (FDU) Fil_lock(intento) podemos intentar bloquear el archivo de base de datos activo mientras se hacen las modificaciones necesarias. El parámetro "intento" controla el número de veces con que se intentara bloquear el archivo.

Utilizando la FDU Rec_lock(intento) podemos intentar bloquear un registro mientras se hacen las modificaciones necesarias y debemos especificar el número de veces con que se intentará bloquear el registro. El registro que queda bloqueado va a ser el que esta apuntando el apuntador de registros.

Por ultimo la función de usuario Add_rec(intento) intentar agregar un registro en blanco a la base de datos activa. El parámetro sirve para especificar el número de veces con que se intentará agregar el registro en blanco.

Finalmente se da un código fuente como ejemplo, para la creación de programas multiusuario. En este ejemplo se pueden dar altas, bajas, consultas y reportes de *ordenes de trabajo.*

Los códigos fuentes de las funciones Zoom(), Zoom_inv(), Nota(), BorRen(), Muevelzq(), Procesa() y Mensajes() no se agregaron debido a que pertenecen a las librerías ArLib y a TexLib, las cuales están hechas en leguaje ensamblador y lenguaje C. Estás funciones son para crear ventanas, borrar ventanas, enviar mensajes, desplazar mensajes, etc.

```
/*
 PROGRAMA:      LAN.PRG
 COMPAÑIA:      Instituto Mexicano Del Petróleo
 DESARROLLO:    Fernando Tecuapacho Texis
 FECHA:         Marzo/1992
 LUGAR:         México D.F.
 OBJETIVO:      Mostrar como se deben usar las funciones de Clipper y las
                funciones de usuario para trabajar con archivos de bases
                de datos en una RED de AREA LOCAL.
 TIPO DE SOFTWARE UTILIZADO : Clipper 5.01, Librería ArLib y TexLib.
*/
#INCLUDE "INKEY.CH"

FUNCTION main_red()
 LOCAL bandera:=.t., file:=[t:\o_t1], c_ren22
 SET DELETED ON
 SET MESSAGE TO 22 CENTER
 SET DATE Ital
 SET WRAP On
 SET SCOR Off
 SETCURSOR(2)
 Cls
 LogoImp()
 Zoom(1,0,22,79,[/w+],SPACE(9),[],.f.,[])
 Zoom(0,0,03,79,[g+/b+],[simple],[],.f.,[])
 Zoom(22,0,22,79,[g+/b+],[simple],[],.f.,[])
 SETCOLOR([bg+/b+])
 @01,71 SAY Time()
 @02,71 SAY DATE()
 SETCOLOR([+w/br+])
 @02,01 SAY [-F2-]
 @02,06 SAY [AYUDA]
 SETCOLOR([+w/b+])
Nota([IMP - ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES - U.N.A.M.],1)
Nota([A R A G O N],2)
SET KEY K_F2 TO AYUDA

 IF ! Net_use(file,.F.,3)
   Zoom_inv(0,0,24,79,[w/n],SPACE(8))   //El archivo esta abierto en forma EXCLUSIVA
   RETURN
 END
 IF ! FILE([t:\o_t1.ntx])                //Se determina si existe el archivo .ntx
   INDEX ON ot TO t:\o_t1                //Se crea archivo índice
```

```
ELSE
  SET INDEX TO t:\o_t1                    //Se pone activo al archivo índice
END
c_ren22:= SAVESCREEN(22,0,22,79)
WHILE bandera                             //Ciclo de opciones
 SETCOLOR([bg+/b+])
 @01,71 SAY Time()
 Zoom(4,1,13,18,"g+/b+",[simple],7,.t.,"RB")
 SETCOLOR([w+/b+,w+/r+])
 @05,3 PROMPT [ALTAS]          MESSAGE [Agregar un nuevo consecutivo a la O.T.]
 @06,3 PROMPT [BAJAS]          MESSAGE [Dar de baja una O. T.]
 @07,3 PROMPT [CONSULTAS IND.] MESSAGE [Consulta las diferentes O. T.]
 @08,3 PROMPT [TABLA DE O. T.] MESSAGE [Consulta las diferentes O. T.]
 @09,3 PROMPT [MODIFICACIONES] MESSAGE [Permite modificar la descripción de;
 la O.T.]
 @10,3 PROMPT [REPORTES]   MESSAGE [Módulo para imprimir O. T.]
 @11,3 PROMPT [UTILERIAS]  MESSAGE [Módulo para depurar bases de datos]
 @12,3 PROMPT [SALIR       ]  MESSAGE [Regresar a MS-DOS]
 MENU TO opcion
 RESTSCREEN(22,0,22,79,c_ren22)
 SETCOLOR([w+/n])
 BorRen(23)
 DO CASE
   CASE opcion==1
      Red_altas()
   CASE opcion==2
      Red_bajas()
   CASE opcion==3
      Red_consu()
   CASE opcion==4
      Red_con_t()
   CASE opcion==5
      Red_modif()
   CASE opcion==6
      Red_Repor()
   CASE opcion==7
      Red_utile()
   CASE opcion==8 .OR. LASTKEY()=K_ESC
      Red_salir()
  ENDCASE
ENDDO
RETURN(NIL)
*
```

```
FUNCTION Red_salir()       //Cierra todos los archivos y retorna el control al MS-DOS
 Close all
 Zoom_inv(0,0,24,79,[w/n],SPACE(8))
 QUIT
RETURN(NIL)
*

/*
 PROGRAMA:        Red_altas()
 COMPAÑIA:        Instituto Mexicano Del Petróleo
 DESARROLLO:      Fernando Tecuapacho Texis
 FECHA:           Marzo/1994
 LUGAR:           México D.F.
 OBJETIVO:        Crear ordenes de trabajo en una RED de AREA LOCAL.
                  Las ordenes de trabajo se generan en forma consecutiva.
 TIPO DE SOFTWARE UTILIZADO : Clipper 5.01, Librería ArLib y TexLib.
*/
FUNCTION Red_altas()
 LOCAL bandera:=.f.
 LOCAL c_ot:=SPACE(LEN(ot)) ,c_proy:=SPACE(LEN(proy)),
c_oficio:=SPACE(LEN(oficio)),;
     c_d_d:=SPACE(LEN(d_d)), c_desc:=SPACE(LEN(desc)),
c_resp:=SPACE(LEN(resp)),;
     n_mon_fac:=0
 c_ot:=SUBSTR(STR(VAL(ot) + 1),3,8)
 Zoom(8,22,20,70,"g+/b+",[simple],7,.t.,"RB")
 SETCOLOR([gr+/b+,w+/r+])
 @09,24 SAY [PROYECTO    ] GET c_proy   PICT[@!]
 @09,55 SAY [O. T.]
 @11,24 SAY [OFICIO      ] GET c_oficio  PICT[@!]
 @13,24 SAY [DEPARTAMENTO ] GET c_d_d    PICT[@!]
 @13,52 SAY [RESPONSABLE  ] GET c_resp   PICT[@!]
 @15,24 SAY [DESCRIPCION  ] GET c_desc   PICT [@S31]
 @17,24 SAY [COSTO DE LA FACTURA ] GET n_mon_fac  PICT[999,999,999]
 READ
 SETCOLOR([w+/n+])
 IF ! Fil_lock(5) .OR. LASTKEY()==27      //Se determina si el archivo no está bloqueado
   MUEVEIZQ2(20,25,[ No se le puede asinar una O. T. en este momento],[gr+/bg+])
   Zoom_inv(8,20,21,72,[/w+],SPACE(8))
   RETURN(NIL)  //Archivo bloqueado por otro usuario
 END
```

```
GO BOTTOM
c_ot:=SUBSTR(STR(VAL(ot) + 1),3,8)
IF Add_rec(2)          //Se habre un registro en blanco y se remplaza la información.
  REPLACE ot      WITH c_ot
  REPLACE proy    WITH c_proy
  REPLACE oficio  WITH c_oficio
  REPLACE d_d     WITH c_d_d
  REPLACE desc    WITH c_desc
  REPLACE resp    WITH c_resp
  REPLACE mon_fac WITH n_mon_fac
  COMMIT
  UNLOCK
  SETCOLOR([w+/r+])
  @09,61 SAY c_ot PICT[@!]
  MUEVEIZQ2(20,25,[    Su ORDEN DE TRABAJO es la ] + c_ot,[gr+/bg+])
ELSE                   //El archivo está bloqueado por otro usuario
  UNLOCK
  MUEVEIZQ2(20,20,[ No sele puede asinar una O. T. en este momento],[gr+/bg+])
END
Zoom_inv(8,22,21,72,[/w+],SPACE(8))
RETURN(NIL)
*

/*
 PROGRAMA:        Red_bajas()
 COMPAÑIA:        Instituto Mexicano Del Petróleo
 DESARROLLO:      Fernando Tecuapacho Texis
 FECHA:           Marzo/1994
 LUGAR:           México D.F.
 OBJETIVO:        Se dan de baja ordenes de trabajo. Unicamente quedan marcados
                  para ser eliminados posteriormente.
 TIPO DE SOFTWARE UTILIZADO : Clipper 5.01, Librería ArLib y TexLib.
*/
FUNCTION Red_bajas()
 LOCAL c_ot:=SPACE(LEN(OT)), c_proy, c_podi, c_oficio, c_d_d, c_desc
 Zoom(6,22,20,70,"g+/b+",[simple],7,.t.,"RB")
 SETCOLOR([gr+/b+,w+/r+])
 @07,24 SAY [O. T. QUE DESEA DAR DE BAJA ] GET c_ot
 READ
 SETCOLOR([w+/n])
 SEEK c_ot
```

```
IF FOUND()
  IF ! Rec_lock(5)
    UNLOCK
    Zoom_inv(6,22,21,72,|/w+|,SPACE(8))
    RETURN(NIL)  //El registro está bloqueado por otro usuario
  END
ELSE
  Nota(|No existe esa orden de trabajo...|,23)
  INKEY(0)
  BorRen(23,|w+/n|)
  Zoom_inv(6,22,21,72,|/w+|,SPACE(8))
  RETURN(NIL)
END
SEEK c_ot     //Se vuelve a buscar el registro por si estaba bloqueado
IF ! FOUND()
  UNLOCK
  MUEVEIZQ2(22,20,|   Fue borrada la O. T. por otro usuario...|,|gr+/bg+|)
  Zoom_inv(6,22,21,72,|/w+|,SPACE(8))
  RETURN(NIL)
END
c_proy:=proy; c_oficio:=oficio; c_d_d:=d_d; c_desc:=desc; c_resp:=resp;
n_mon_fac:=mon_fac
SETCOLOR(|gr+/b+|)
@09,24 SAY |PROYECTO    | + c_proy   PICT|@!|
@11,24 SAY |OFICIO      | + c_oficio  PICT|@!|
@13,24 SAY |DEPARTAMENTO | + c_d_d    PICT|@!|
@13,52 SAY |RESPONSABLE  | + c_resp   PICT|@!|
@15,24 SAY |DESCRIPCION  | + c_desc   PICT |@S31|
@17,24 SAY |COSTO DE LA FACTURA |
@17,45 SAY n_mon_fac  PICT|999,999,999|
SETCOLOR(|bg+/b+|)
@19,35 SAY |Seguro de eliminar la O. T. <S/N>|
IF UPPER(CHR(INKEY(0)))= |S|
  DELETE
END
UNLOCK
COMMIT
Zoom_inv(6,22,21,72,|/w+|,SPACE(8))
RETURN(NIL)
*
```

```
/*
 PROGRAMA:       Red_consu()
 COMPAÑIA:       Instituto Mexicano Del Petróleo
 DESARROLLO:     Fernando Tecuapacho Texis
 FECHA:          Marzo/1994
 LUGAR:          México D.F.
 OBJETIVO:       Consultar ordenes de trabajo individuales en la RED de AREA
LOCAL.
 TIPO DE SOFTWARE UTILIZADO : Clipper 5.01, Librería ArLib y TexLib.
*/

FUNCTION Red_consu()
 LOCAL c_ot:=SPACE(LEN(OT)), c_proy, c_podi, c_oficio, c_d_d, c_desc
 Zoom(6,22,20,70,"g+/b+",[simple],7,.t.,"RB")
 SETCOLOR([gr+/b+,w+/r+])
 @07,24 SAY [O. T. QUE DESEA DAR CONSULTAR] GET c_ot
 READ
 SETCOLOR([w+/n])
 SEEK c_ot
 IF ! FOUND()
   Nota([No existe esa orden de trabajo...],23)
   INKEY(0)
   BorRen(23,[w+/n])
   Zoom_inv(6,22,21,72,[/w+],SPACE(8))
   RETURN(NIL)
 END
 c_proy:=proy; c_oficio:=oficio; c_d_d:=d_d; c_desc:=desc; c_resp:=resp;
n_mon_fac:=mon_fac
 SETCOLOR([gr+/b+])
 @09,24 SAY [PROYECTO     ] + c_proy   PICT[@!]
 @11,24 SAY [OFICIO       ] + c_oficio PICT[@!]
 @13,24 SAY [DEPARTAMENTO ] + c_d_d    PICT[@!]
 @13,52 SAY [RESPONSABLE  ] + c_resp   PICT[@!]
 @15,24 SAY [DESCRIPCION  ] + c_desc   PICT [@S31]
 @17,24 SAY [COSTO DE LA FACTURA ]
 @17,45 SAY n_mon_fac  PICT[999,999,999]
 SETCOLOR([w+/n])
 Nota([Presione una tecla para continuar...],23)
 INKEY(0)
 BorRen(23,[w+/n])
 Zoom_inv(6,22,21,72,[/w+],SPACE(8))
RETURN(NIL)
*
```

```
/*
 PROGRAMA:      Red_con_t()
 COMPAÑIA:      Instituto Mexicano Del Petróleo
 DESARROLLO:    Fernando Tecuapacho Texis
 FECHA:         Marzo/1994
 LUGAR:         México D.F.
 OBJETIVO:      Consultar las ordenes de trabajo, visualizando todo el catalogo en
                una RED de AREA LOCAL.
 TIPO DE SOFTWARE UTILIZADO : Clipper 5.01, Librería ArLib y TexLib.
*/
FUNCTION Red_con_t()
 Mensajes(1)
 SETCOLOR([gr+/bg+,w+/r+])
 DBGOTOP()
 BROWSE(6,22,21,77)
 BorRen(24,[w+/n])
RETURN(NIL)
*

/*
 PROGRAMA:      Red_modi()
 COMPAÑIA:      Instituto Mexicano Del Petróleo
 DESARROLLO:    Fernando Tecuapacho Texis
 FECHA:         Marzo/1994
 LUGAR:         México D.F.
 OBJETIVO:      Hacer modificaciones a las ordenes de trabajo con el archivo de bases
                de datos en una RED de AREA LOCAL.
 TIPO DE SOFTWARE UTILIZADO : Clipper 5.01, Librería ArLib y TexLib.
*/
FUNCTION Red_modif()
 LOCAL c_ot:=SPACE(LEN(OT)), c_proy, c_podi, c_oficio, c_d_d, c_desc
 Zoom(6,22,20,70,"g+/b+",[simple],7,.t.,"RB")
 SETCOLOR([gr+/b+,w+/r+])
 @07,24 SAY [O. T. QUE DESEA MODIFICAR] GET c_ot
 READ
 SETCOLOR([w+/n])
 SEEK c_OT
 IF FOUND()
   IF ! Rec_lock(5)
     Zoom_inv(6,22,21,72,[/w+],SPACE(8))
     RETURN(NIL)  //El registro esta bloqueado por otro usuario
   END
 ELSE
```

```
   Nota([No existe esa orden de trabajo...],23)
   INKEY(0)
   BorRen(23,[w+/n])
   Zoom_inv(6,22,21,72,[/w+],SPACE(8))
   RETURN(NIL)
 END
 c_proy:=proy; c_oficio:=oficio; c_d_d:=d_d; c_desc:=desc; c_resp:=resp;
n_mon_fac:=mon_fac
 SETCOLOR([gr+/b+,w+/r+])
 @09,24 SAY [PROYECTO     ] GET c_proy    PICT[@!]
 @09,55 SAY [O. T.]
 @11,24 SAY [OFICIO       ] GET c_oficio  PICT[@!]
 @13,24 SAY [DEPARTAMENTO ] GET c_d_d     PICT[@!]
 @13,52 SAY [RESPONSABLE  ] GET c_resp    PICT[@!]
 @15,24 SAY [DESCRIPCION  ] GET c_desc    PICT [@S31]
 @17,24 SAY [COSTO DE LA FACTURA ] GET n_mon_fac  PICT[999,999,999]
 READ
 REPLACE ot      WITH c_ot
 REPLACE proy    WITH c_proy
 REPLACE oficio  WITH c_oficio
 REPLACE d_d     WITH c_d_d
 REPLACE desc    WITH c_desc
 REPLACE resp    WITH c_resp
 REPLACE mon_fac WITH n_mon_fac
 COMMIT
 UNLOCK
 Zoom_inv(6,22,21,72,[/w+],SPACE(8))
 RETURN(NIL)
 *
```

```
/*
 PROGRAMA:       Red_Report()
 COMPAÑIA:       Instituto Mexicano Del Petróleo
 DESARROLLO:     Fernando Tecuapacho Texis
 FECHA:          Marzo/1994
 LUGAR:          México D.F.
 OBJETIVO:       Imprimir la información de la orden de trabajo que se indique.
                 La impresión puede direccionarse al puerto local o la impresora
                 que está conectada en el servidor de la RED de AREA LOCAL.
 TIPO DE SOFTWARE UTILIZADO : Clipper 5.01, Librería ArLib y TexLib.
*/
FUNCTION Red_Repor()
 LOCAL c_ot:=SPACE(LEN(OT)), c_proy, c_podi, c_oficio, c_d_d, c_desc
 Zoom(6,22,8,70,"g+/b+",[simple],7,.t.,"RB")
 SETCOLOR([gr+/b+,w+/r+])
 @07,24 SAY [O. T. QUE DESEA IMPRIMIR] GET c_ot
 READ
 SETCOLOR([w+/n])
 SEEK c_ot
 IF ! FOUND()
   Nota([No existe esa orden de trabajo...],23)
   INKEY(0)
   BorRen(23,[w+/n])
   Zoom_inv(6,22,9,72,[/w+],SPACE(8))
   RETURN(NIL)
 END
 c_proy:=proy; c_oficio:=oficio; c_d_d:=d_d; c_desc:=desc; c_resp:=resp;
n_mon_fac:=mon_fac
 Nota([Impresión en la impresora local. ( S/N )],23)
 IF UPPER(CHR(INKEY(0))) == [N]          //Determina cual impresora se usará.
    SET PRINTER TO LPT2
 ELSE
    SET PRINTER TO LPT1
 ENDIF
 BorRen(23,[w+/n])
 Nota([Espere imprimiendo registro...],23)
 SET DEVICE TO PRINTER
 @09,24 SAY [PROYECTO] + c_proy   PICT[@!]
 @11,24 SAY [OFICIO     ] + c_oficio  PICT[@!]
 @13,24 SAY [DEPARTAMENTO ] + c_d_d    PICT[@!]
 @13,52 SAY [RESPONSABLE  ] + c_resp   PICT[@!]
 @15,24 SAY [DESCRIPCION  ] + c_desc   PICT [@S31]
 @17,24 SAY [COSTO DE LA FACTURA ]
```

```
@17,45 SAY n_mon_fac  PICT[999,999,999]
EJECT
SET PRINTER TO
SET DEVICE TO SCREEN
Zoom_inv(6,22,9,72,[/w+],SPACE(8))
RETURN(NIL)
*

/*
 PROGRAMA:       Red_utile()
 COMPAÑIA:       Instituto Mexicano Del Petróleo
 DESARROLLO:     Fernando Tecuapacho Texis
 FECHA:          Marzo/1994
 LUGAR:          México D.F.
 OBJETIVO:       Borra los registros marcados. Habriendo los archivos en
                 forma exclusiva.
 TIPO DE SOFTWARE UTILIZADO : Clipper 5.01, Librería ArLib y TexLib.
*/
FUNCTION Red_utile()
 LOCAL file:=[t:\o_t1]
 CLOSE ALL
 IF ! Net_use(file,.t.,5)
   Net_use(file,.f.,5)  //No se pudo abrir en forma EXCLUSIVA
   Nota([No se lograrón depurar los archivos...],23)
   INKEY(5)
 ELSE
   IF ! FILE([t:\o_t1.ntx])
     INDEX ON ot TO t:\o_t1
   ELSE
     SET INDEX TO t:\o_t1
   END
   Nota([Espere depurando archivos...],23)
   PACK;   CLOSE ALL
   WHILE ! Net_use(file,.f.,5)/       /Hasta que se logre abrir en forma exclusiva
   END
 END
 IF ! FILE([t:\o_t1.ntx])
   INDEX ON ot TO t:\o_t1
 ELSE
   SET INDEX TO t:\o_t1
 END
 BorRen(23,[w+/n])
RETURN(NIL)
```

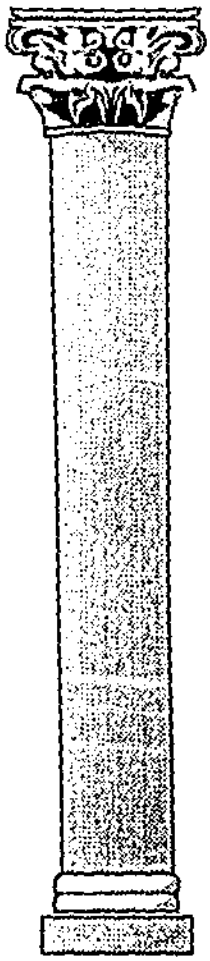
```
/*
 PROGRAMA:      Net_use()
 COMPAÑIA:      Instituto Mexicano Del Petróleo
 DESARROLLO:    Fernando Tecuapacho Texis
 FECHA:         Marzo/1994
 LUGAR:         México D.F.
 OBJETIVO:      Intentar abrir la base de datos enviada en el parámetro "file" con
                el modo definido en la variable "ex_use" y con el número de veces
                controlado con la variable "intento".
 TIPO DE SOFTWARE UTILIZADO : Clipper 5.01, Librería ArLib y TexLib.
*/
FUNCTION Net_use(file,ex_use,intento)
 LOCAL contador, i, n_cursor:=SETCURSOR()
 SETCURSOR(0)
 contador:= intento
 WHILE (contador > 0) .AND. LASTKEY()#K_ESC
   IF ex_use
     USE (file) EXCLUSIVE NEW    //abrir en forma exclusiva
   ELSE
     USE (file) SHARED NEW       //abrir en forma compartida
   END
   IF .NOT. NETERR()
     SETCURSOR(n_cursor)
     RETURN(.T.)                 //Archivo disponible
   ELSE
     Si([Archivo ocupado, ¿Continúo?],24,[w+/n,w+/r])
     IF S=[N]
       EXIT                      //El archivo está bloqueado por otro usuario
     ELSE
       BorRen(23)
       Nota([Espere intentando abrir archivo...],23)
       Procesa(23,58,10)
       BorRen(23)
       contador--
     ENDIF
   END
 ENDDO
 SETCURSOR(n_cursor)
 RETURN(.F.)                     // Archivo no disponible
*
```

```
/*
 PROGRAMA:       Fil_lock()
 COMPAÑIA:       Instituto Mexicano Del Petróleo
 DESARROLLO:     Fernando Tecuapacho Texis
 FECHA:          Marzo/1994
 LUGAR:          México D.F.
 OBJETIVO:       Intentar bloquear la base de datos activa, el parámetro "intento".
                 Controla el número de veces que se intentara bloquer el archvio.
 TIPO DE SOFTWARE UTILIZADO : Clipper 5.01, Librería ArLib y TexLib.
*/
FUNCTION Fil_lock(intento)
LOCAL siempre, n_cursor:=SETCURSOR()
SETCURSOR(0)
IF FLOCK()
  SETCURSOR(n_cursor)
  RETURN(.T.)                    //Bloqueo del achivo aceptado
END
contador:= intento
WHILE contador > 0
  IF FLOCK()
    SETCURSOR(n_cursor)
    RETURN(.T.)
  END
  Nota([Archivo ocupado, ¿Continúo? ( S/N )],23)
  IF UPPER(CHR(INKEY(0))) == [N]
    EXIT                         // El archivo está bloqueado por otro usuario
  ELSE
    Nota([Espere intentando abrir archivo o registro...],23)
    Procesa(23,64,10)
    BorRen(23,[w+/n])
    contador--
  ENDIF
ENDDO
SETCURSOR(n_cursor)
BorRen(23,[w+/n])
RETURN(.F.)                      // El archivo está bloqueado por otro usuario
*
```

```
/*
 PROGRAMA:      Add_rec()
 COMPAÑIA:      Instituto Mexicano Del Petróleo
 DESARROLLO:    Fernando Tecuapacho Texis
 FECHA:         Marzo/1994
 LUGAR:         México D.F.
 OBJETIVO:      Intentar agregar un registro a la base de datos activa y como
                parámetro el número de intenos controlado con la variable "intento".
 TIPO DE SOFTWARE UTILIZADO : Clipper 5.01, Librería ArLib y TexLib.
*/
FUNCTION Add_rec(intento)
 LOCAL contador,n_cursor:=SETCURSOR()
 SETCURSOR(0)
 APPEND BLANK
 IF .NOT. NETERR()
   SETCURSOR(n_cursor)
   RETURN(.T.)                    //Se puede agregar un nuevo registro
 END
 contador:= intento
 WHILE contador > 0
   IF .NOT. NETERR()
     SETCURSOR(n_cursor)
     RETURN(.T.)                  //Se puede agregar un nuevo registro
   END
   Nota([Archivo ocupado, ¿Continúo? ( S/N )],23)
   IF UPPER(CHR(INKEY(0))) == [N]
     EXIT                         //No se puede agregar un nuevo registro
   ELSE
     BorRen(23)
     Nota([Espere intentando agregar el nuevo registro...],23)
     Procesa(23,58,10)
     BorRen(23)
     contador --
   ENDIF
 ENDDO
 SETCURSOR(n_cursor)
RETURN(.F.)                       //No se puede agregar un nuevo registro
*
```

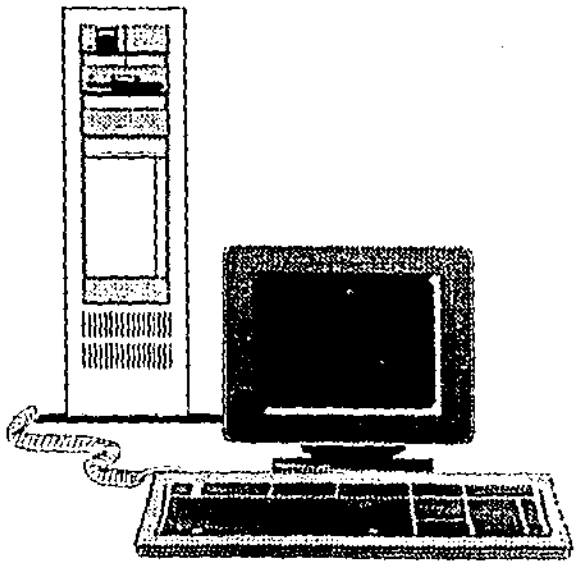
```
/*
 PROGRAMA:        Fil_lock()
 COMPAÑIA:        Instituto Mexicano Del Petróleo
 DESARROLLO:      Fernando Tecuapacho Texis
 FECHA:           Marzo/1994
 LUGAR:           México D.F.
 OBJETIVO:        Intentar bloquear un registro de la base de datos activa, el parámetro
                  "intento" controla el número de veces que se intentara bloquear el
                  registro.
 TIPO DE SOFTWARE UTILIZADO : Clipper 5.01, Librería ArLib y TexLib.
*/
FUNCTION Rec_lock(intento)
 LOCAL contador, n_cursor:=SETCURSOR()
 SETCURSOR(0)
 IF RLOCK()
   SETCURSOR(n_cursor)
   RETURN(.T.)    // El registro está disponible
 END
 contador:= intento
 WHILE contador > 0
   IF RLOCK()
     SETCURSOR(n_cursor)
     RETURN(.T.)  // El registro está disponible
   END
   contador --
   Nota([Registro ocupado, ¿Continúo? ( S/N )],23)
   IF UPPER(CHR(INKEY(0))) == [N]
     EXIT  // El archivo o registro está bloqueado por otro usuario
   ELSE
     BorRen(23)
     Nota([Espere intentando abrir archivo o registro...],23)
     Procesa(23,64,10)
     BorRen(23)
     contador --
   ENDIF
 ENDDO
 SETCURSOR(n_cursor)
 BorRen(23)
RETURN(.F.)  // El registro no está disponible
*
```

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO



CAPITULO 10

- INTERFASE CON LENGUAJE C Y ENSAMBLADOR



ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES
ARAGON

10.1 EL SISTEMA EXTENDIDO DE CLIPPER

El sistema extendido es un grupo de funciones y macros para Lenguaje C y ensamblador que permiten crear y usar Funciones Definidas por el Usuario (UDF) en dichos lenguajes. Para hacer uso de él es necesario conocer las generalidades del encadenamiento, paso de parámetros, manejo de memoria, etc.

Al programar en Lenguaje C o ensamblador, puede nombrar a sus funciones con 10 caracteres o menos. Para nombrarlas puede utilizar mayúsculas ó minúsculas, pero durante la compilación se transforman a mayúsculas.

Una vez que han sido compiladas y encadenadas, la sintaxis de uso para dichas funciones es la misma que para las UDF hechas en Clipper.

¿ Porque usar Lenguaje C y Ensamblador ?

Clipper proporciona un lenguaje poderoso para la creación de aplicaciones relacionadas con el manejo de Bases de Datos, pero de ninguna manera es un lenguaje de propósito general.

Clipper 5.01 fue creado con Microsoft C 5.1 y muchas de la funciones de la librería de C no están disponibles en Clipper. Por lo tanto existen ciertas tareas que realizarlas en Clipper resulta poco práctico o imposible. Además hay ciertas operaciones que se desempeñan en una forma más eficiente en C, tales como la manipulación de cadenas y cálculos matemáticos.

En general, los programas hechos en lenguaje ensamblador se ejecutan más rápido que los programas hechos en otros lenguajes y además son más pequeños. El lenguaje ensamblador es usado para accesar funciones del DOS y tener control directo de las operaciones de entrada y salida, mediante el acceso a las funciones del BIOS (Binary Input Output System).

10.1.1 Las funciones y macros del sistema extendido

Clipper proporciona algunas funciones y macros para facilitar la comunicación con lenguaje C Y Ensamblador.

El sistema extendido proporciona las siguientes funciones:

* Funciones para recibir parámetros provenientes de Clipper.
* Funciones para retornar parámetros por valor a Clipper.
* Funciones para retornar parámetros por referencia a Clipper.
* Funciones para manejo de memoria

Funciones para recibir parámetros provenientes de Clipper:

_parc()	Obtiene un parámetro de tipo "char"
_parclen()	Obtiene la longitud de un parámetro de tipo "char"
_pards()	Obtiene un parámetro tipo fecha como tipo "string"
_parinfa()	Obtiene el tipo de dato de un elemento de un arreglo pasado como parámetro
_parinfo()	Obtiene el tipo de dato de un parámetro
_parl()	Obtiene un parámetro lógico como tipo int
_parnd()	Obtiene un parámetro numérico como tipo "double"
_parni()	Obtiene un parámetro numérico como tipo "int"
_parnl()	Obtiene un parámetro numérico como tipo "long"

Funciones para enviar parámetros por valor a Clipper:

_ret()	Retorna un valor "NIL"
_retc()	Retorna una cadena terminada.
_retclen()	Retorna la longitud de una cadena.
_retds()	Retorna una cadena que contiene una fecha
_retl()	Retorna un entero como tipo lógico
_retnd()	Retorna un valor numérico como tipo "double"
_retni()	Retorna un valor numérico como tipo "int"
_retnl()	Retorna un valor numérico como tipo "long"

Funciones para enviar parámetros por referencia a Clipper:

_storc()	Asigna a una variable una cadena terminada con un nulo
_storclen()	Asigna a una variable la longitud de una cadena.
_stords()	Asigna un valor tipo fecha a una variable, mediante una cadena.
_storl()	Asigna un valor lógico a una variable
_stornd()	Asigna un valor de tipo numérico a una variable usando un valor tipo "double"
_storni()	Asigna un valor numérico como "int"
_stornl()	Asigna un valor numérico como "long"

Funciones para manejo de memoria

_xalloc()	Reserva memoria y retorna NULL si no tiene éxito
_xfree()	Libera una localidad de memoria reservada anteriormente.
_xgrab()	Reserva memoria, generando un error si no tiene éxito

Las macros

El archivo EXTEND.H proporciona las siguientes constantes manifiestas (int representa un entero) :

ALENGHT(int)	Indica el número de elementos del arreglo pasado como parámetro
ISARRAY(int)	Determina si el parámetro pasado es un arreglo
ISBYREF(int)	Determina si el parámetro pasado es por referencia
ISCHAR(int)	Determina si el parámetro pasado es un caracter
ISDATE(int)	Determina si el parámetro pasado es una fecha
ISLOG(int)	Determina si el parámetro pasado es de tipo lógico
ISMEMO(int)	Determina si el parámetro pasado es tipo memo
ISNUM(int)	Determina si el parámetro pasado es numérico
PCOUNT	Indica el número de parámetros pasados

10.1.2 Obtención y retorno de parámetros

El puente de comunicación de Clipper a Lenguaje C y ensamblador es el paso y retorno de parámetros, por valor y por referencia.

Obtención de parámetros

En la tabla 10.1 se puede observar como se obtiene en los programas de C y ensamblador cada parámetro enviado desde Clipper. Por ejemplo, las fechas son pasadas como cadenas de caracteres y los números son pasados en la manera apropiada para C y ensamblador.

Note que cada función acepta dos parámetros. El primero da la posición en la lista. Por ejemplo, si en Lenguaje C queremos obtener el tercer parámetro pasado y es de tipo caracter, se debe usar:

```
_parc(3)
```

El segundo parámetro habilita el paso de parámetros en un arreglo. Si en el ejemplo anterior el tercer elemento pasado es un arreglo y queremos obtener el quinto elemento de ese arreglo se debe usar:

```
_parc(3,5)
```

CLIPPER	TIPO DE DATO	LENGUAJE C	ENSAMBLADOR
character	char*	_parc(int[,int])	_PARC
date	char*	_pards(int[,int])	_PARDS
logical	int	_parl(int[,int])	_PARL
numeric	int	_parni(int[,int])	_PARNI
numeric	long	_parnl(int[,int])	_PARNL
numeric	double	_parnd(int[,int])	_PARDND

Tabla 10.1 Recepción de parámetros de Clipper

Retorno de parámetros por valor

En las UDF sólo un parámetro puede ser retornado a Clipper. Existe una función de retorno para cada tipo de dato. Estas funciones se muestran en la tabla 10.2.

CLIPPER	LENGUAJE C	ENSAMBLADOR
character	_retc(char*])	_RETC
date	_retds(char*])	_RETDS
logical	_retl(int)	_RETL
numeric	_retni(int)	_RETNI
numeric	_retnl(long)	_RETNL
numeric	_retnd(double)	_RETDND
	_ret()	_RET
	retclen(char*,int)	_RETCLEN

Tabla 10.2 Retorno de parámetros, por valor, a Clipper

Retorno de parámetros por referencia

Es muy útil el retorno de valores por referencia cuando lo que se necesita retornar a Clipper es más de un valor o cuando la variable de retorno ocupa un bloque de memoria muy grande. Estas funciones se muestran en la tabla 10.3

CLIPPER	NOMBRE	ENSAMBLADOR
character	_storc(char*,int[,int])	_STORC
date	_stords(char*,int[,int])	_STORDS
logical	_storl(int,int[,int])	_STORL
numeric	_storni(int,int[,int])	_STORNI
numeric	_stornl(long,int[,int])	_STORNL
numeric	_stornd(double,int[,int])	_STORDND

Tabla 10.3 Retorno de parámetros, por referencia, a Clipper

10.1.3 Compilación y encadenado con lenguaje C y ensamblador

El llamar funciones de C y ensamblador desde Clipper involucra el encadenar dos o más archivos objeto que contienen el código compilado. Uno o algunos de ellos serán archivos objeto de Clipper, los demás serán archivos objeto de C y/o ensamblador

La compilación de los archivos fuente de Clipper se realiza normalmente:

CLIPPER <Arch1Clipper> [,<Arch2Clipper>...,<ArchNClipper>]

La compilación de los archivos fuente de C y ensamblador se explica a continuación

10.1.3.1 Compilación y encadenado con Lenguaje C

Para compilar una función creada con Microsoft C, se deben incluir las siguientes opciones:

CL /c /AL /FPa /Gs /Oalt /ZL <Archivo.C>

Donde <Archivo.C> es el nombre del archivo fuente de C. Las opciones se explican a continuación:

/c Compila sin encadenar.
/AL Compila el archivo con el modelo de memoria largo.
/FPa Especifica la librería de punto flotante alternativa.
/Gs Omite la llamada al stack para checar las rutinas.
/Oalt Establece la optimización para la compilación.
/Zl Indica al compilador que no incluya el nombre de la librería en el archivo objeto.

Al especificar las opciones recuerde que el compilador de C distingue entre mayúsculas y minúsculas.

Para realizar el encadenado y obtener un archivo .EXE se debe realizar el encadenado de la siguiente manera

RTLINK FI <ArchClipper.obj> [,<ArchClipper.obj>,...], <ArchC.obj> [,<ArchC.obj>,...] <LIB> <Librerías>

Donde <Librerías> representa los nombres de las librerías necesarias para realizar el encadenado

10.1.3.2 Construcción de librerías con Lenguaje C

Una técnica para hacer la interfase con lenguaje C es la creación de librerías hechas a base de programas objeto de Microsoft C.

Podemos tener en una librería varios módulos objeto y al momento del encadenado sólo se añadirán a la aplicación aquellos módulos que necesite. RTLINK llamará a estas librerías, de la misma forma en que llama a las librerías proporcionadas por Clipper

Para crear una librería de C se requieren los siguientes pasos:

* Compilar los archivos de C de la siguiente manera:
  CL /c /AL /FPa /Gs /Oalt /ZL <Archivo.C>

* Ejecutar el programa LIB:EXE proporcionado por Microsoft C con la siguiente sintaxis:

  LIB <NombLibr> [Opc] <ArchC.obj> [,[Opc] <ArchC.obj>,...]

  <NombLibr> Es el nombre de la librería
  <ArchC.obj> Es un archivo objeto
  <Opc> Es una de las opciones que se explican abajo:

  + módulo Añade un módulo a nuestra librería. Un módulo puede ser un programa objeto, pero también puede ser una librería. Se asume por defecto la extensión .OBJ

  - módulo Borra el módulo especificado de la librería.

-+ módulo Reemplaza un módulo existente en la librería por otro del mismo nombre.

* módulo Copia el módulo indicado a un archivo .OBJ sin borrarlo de la librería

-* módulo Copia el módulo indicado a un archivo .OBJ borrándolo de la librería

10.1.3.3 Compilación y encadenado con Lenguaje Ensamblador

Para compilar la rutina en ensamblador se usa la siguiente sintaxis:

MASM <ArchivoEnsamblador>;

El punto y coma (;) al final de la sentencia es necesario para aceptar los valores por omisión para el nombre del archivo objeto.

Para encadenar la aplicación Clipper con la rutina en ensamblador y obtener un archivo .EXE, se usa la siguiente sintaxis:

RTLINK FI <ArchClipper.obj> [,<ArchClipper.obj>,...],
<ArchEnsamb.obj> [,<ArchEnsamb.obj>,...]

10.2 CREACION DE FUNCIONES EN LENGUAJE C

Podemos dividir en dos partes a la típica UDF escrita en C. La primera parte sirve como área de recepción y envío: recibe los parámetros, checa su validez y los pasa al proceso. Al terminar el proceso retorna el valor a Clipper. La segunda parte es en sí, la tarea a realizar.

Microsoft C agregará una subraya (_) al inicio del nombre de la función a no ser que se escriba la palabra CLIPPER antes del nombre de la función, por ejemplo:

```
CLIPPER Nom_Fun()        // Está función se llamará desde Clipper
                            // como Nom_Func()

Nom_Fun()   // Está función se llamará desde Clipper como
               // _Nom_Fun()
```

A continuación se muestran algunos ejemplos de funciones hechas en Lenguaje C.

10.2.1 Función para crear nombres de archivo únicos

Frecuentemente es necesario crear archivos temporales durante la ejecución de un programa. En un ambiente monousuario esto puede solucionarse fácilmente, pero en un ambiente de red, si no se tiene cuidado esto puede ocasionar problemas si varios usuarios quieren crear sus archivos temporales con el mismo nombre al mismo tiempo.

Es importante asegurarse que se cree un nombre de archivo único para cada usuario y que no exista en el directorio de trabajo. Esto no puede realizarse mediante Clipper, pero usando lenguaje C se soluciona fácilmente.

La función C_NomUni recibe como parámetro de entrada una cadena de dos caracteres y retorna una cadena de ocho caracteres, de los cuales los dos primeros corresponden a la cadena de caracteres enviada por Clipper.

```
/*
  Función:          C_NomUni.C        Hecha en Microsoft C 6.0
  Creación:         08/03/94          Por:  Jorge Castaño
  Ultima revisión:  17/03/94          Por:  Fernando Tecuapacho

  Propósito:              Obtener un nombre de archivo único.
  Compilar:               CL /c /AL /Zl /Oalt /FPa /Gs C_NomUni.C
*/

#include <extend.h>        /*  Archivo proporcionado por Clipper  */
#include <stdio.h>
#include <io.h>
#include <string.h>

char Plantilla[] ="      ";          /*  Plantilla del nombre del archivo  */
char Sufijo[] ="XXXXXX";            /*  Plantilla del sufijo              */
char NombArch[] ="      ";          /*  Variable que contiene el nombre   */
char* CreaTemp( char* Prefijo )
{
 strcpy( Plantilla,Prefijo );              /*  Agrega el prefijo a la plantilla  */
 strcat( Plantilla,Sufijo );               /*  Agrega el sufijo a la plantilla   */
 strcpy( NombArch,mktemp(Plantilla) );     /*  Crea el nombre de archivo         */
 return( NombArch );
}                                          /*  Fin de la función CreaTemp        */

char Entrada[] = "  ";              /*  Variable para el parámetro de entrada  */
CLIPPER C_NomUni()
{
 char* Retorno;
 strcpy( Entrada,_parc(1) );
 Retorno = CreaTemp( Entrada );     /*  Retorna el nombre del archivo     */
 _retc( Retorno );
}                                   /*  Fin de la función C_NomUni        */
```

10.2.2 Función para cambio de colores

Es común que dentro de las aplicaciones el usuario seleccione los colores que desea utilizar, no sólo antes del programa, sino dentro del mismo, como una utilería adicional.

Para el menú que se tiene visible lo único que hay que hacer es redibujarlo con los nuevos colores, ¿ Pero que hacer con las pantallas previamente salvadas con un SAVESCREEN() ?

Normalmente la pantalla consiste de 25 líneas por 80 columnas , arrojando un total 2000 bytes. Cada uno de estos caracteres está asociado a una dirección de memoria, en donde a cada uno se le asignan dos bytes: uno para asignar el código ASCII y otro para asignar el atributo (color). Por lo tanto, el total requerido para almacenar una pantalla completa es de 4000 bytes.

Cuando en Clipper se ejecuta un SAVESCREEN lo que hace es guardar en memoria de video la región de la pantalla indicada, caracter por caracter, a una cadena de caracteres. Un caracter para el código ASCII y otro para el atributo.

Con lo que se concluye que para cambiar el color a una pantalla previamente salvada, basta con cambiar los caracteres pares de la cadena que la contiene.

Se puede realizar una función en Clipper que realice esa tarea, pero su tiempo de ejecución sería muy lento, puesto que procesaría byte por byte. La otra alternativa es realizar dicha función en Lenguaje C.

La función C_ModCol() recibe dos argumentos, la variable de memoria en donde se almacena la región de pantalla salvada a través de SAVESCREEN() y un valor numérico correspondiente al color que desea modificar dicha región.

Esta función regresa la misma ventana salvada a memoria pero con diferente color. En caso de que no se envíen los argumentos en forma adecuada, regresará como resultado NIL.

```
/*
   Programa:          C_ModCol.C        Hecho en Microsoft C 6.0
   Creación:          08/03/94          Por:  Jorge Castaño
   Ultima revisión:   17/03/94          Por:  Fernando Tecuapacho

   Propósito:         Modificar el color de una pantalla previamente
                      salvada a memoria a través de la función SAVESCREEN()
```

```
                        de Clipper. El color se asigna por medio de un valor
                        numérico entre el 0 y el 255.

  Compilar:             CL /c /AL /Zl /Oalt /FPa /Gs C_ModCol.C
*/

#include <extend.h>     /*  Archivo proporcionado por Clipper  */

CLIPPER C_ModCol( void )
{
  char *Pantalla, *Puntero;
  unsigned int Color;
  char Posi;

  /* Validación de argumentos */
  if ( PCOUNT == 2 && ISCHAR( 1 ) && ISNUM( 2 ) )
    {
       Pantalla = _parc( 1 );    /*  Ventana de Clipper    */
       Color = _parni( 2 );      /*  Color                 */
       Puntero = Pantalla;       /*  Salva apuntador       */
       Posi = 0;
       while ( *Pantalla )
             {
             if ( Posi )         /*  Si es atributo, modifica  */
               {
                *Pantalla = Color;
                Posi = 0;
               }
             else Posi = 1;
                Pantalla++;      /*  Avanza apuntador      */
             }                   /*  Fin del ciclo         */
       _retc( Puntero );         /*  Regresa ventana a Clipper */
    }
  else _ret();                   /*  Si los parámetros no son válidos retorna NIL */
}                                /*  Fin de la función     */
```

10.2.3 Programa para ejemplificar el uso de las funciones hechas en Lenguaje C

A continuación se muestra el listado de programa Clipper que comprueba las dos funciones hechas con C.

```
/*
  Programa:          PRUEB101.PRG      Hecho en Clipper
  Creación:          09/03/94          Por:  Jorge Castaño
  Ultima revisión:   17/03/94          Por:  Fernando Tecuapacho

  Propósito:         Ejemplificar el uso de las funciones hechas con
                     Lenguaje C:  C_NomUni.c y C_CamCol.c
  Compilar:          CLIPPER Prueb101 /N/M/W
  Encadenar:
  RTLINK FI Prueb101,C_NomUni,C_ModCol LIB CLIPPER,EXTEND,LLIBC7
  (LLIBC7 es un libreria proporcionada por Microsoft C, las demás las proporciona
  Clipper                                                              */

FUNCTION PRUEB101()
LOCAL Nombre,Region,NvoColor
CLS
Nombre =C_NomUni("TM")//  Obtiene un nombre de archivo único que
                          //  inicie con "TM"
@5,18 SAY "El nombre del archivo único es: " +Nombre
@7,18 SAY "Presione cualquier tecla para continuar "
Region := SAVESCREEN(3,10,9,68)
INKEY(0)
CLS
RESTSCREEN(3,10,9,68,Region)          // Se reestablece el área con su color original
/*  A continuación se reestablece el area guardada en todos los colores posibles  */
FOR NvoColor =1 TO 255
  Region := C_ModCol(Region,NvoColor) //  Cambia el color del área
  RESTSCREEN(14,10,20,68,Region)      //  Se reestablece el área con un nuevo color
NEXT
INKEY(0)
CLS
RETURN NIL     /*     Fin de prueb101.prg      */
```

10.3 CREACION DE FÚNCIONES EN ENSAMBLADOR

La comunicación con lenguaje ensamblador es más compleja que con C, debido al cuidado con el manejo de memoria y periféricos. Un programa en ensamblador controla el microprocesador en su propio lenguaje, sin la ayuda de comprobaciones del compilador.

Para manejar lenguaje ensamblador es necesario conocer la arquitectura del microprocesador, el manejo de sus instrucciones, manejo de interrupciones y el manejo de la memoria

Los requerimientos generales para el desarrollo de funciones en ensamblador que deseen utilizarse desde Clipper, son los siguientes:

1.- Contar con una versión de ensamblador de Microsoft, MASM 5.0 ó superior.

2.- Declarar las funciones como públicas y con modelo de memoria largo "LARGE".

3.- Declarar las direcciones como "FAR" para asegurar un regreso a Clipper a través de un "FAR RETURN".

4.- Asegurarse que los registros DS y SS apunten al grupo DGROUP antes de utilizar cualesquiera de las funciones del Sistema Extendido.

A continuación se muestran algunos ejemplos de funciones hechas en lenguaje ensamblador.

10.3.1 Función para obtener el nombre de la unidad actual.

Frecuentemente durante la ejecución de una aplicación es necesario saber el nombre de la unidad actual, principalmente si se está trabajando en ambiente de red. No existe una instrucción en Clipper que de el nombre de la unidad actual, pero esto puede resolverse creando una función en lenguaje ensamblador.

La función E_Drive() no tiene parámetros de entrada y devuelve el número de la unidad actual: 0 = A, 1 = B, 2 = C y así sucesivamente.

```
; Función :          E_DRIVE.ASM    Hecha en Ensamblador
; Creación:          11/04/94       Por:  Jorge Castaño
; Ultima revisión:   15/04/94       Por:  Fernando Tecuapacho
; Propósito:                        Retornar el nombre de la unidad actual a un
;                                   programa Clipper
; Sintaxis de uso:                  E_DRIVE()  (No tiene parámetros de entrada)
; Retorna:                          Un valor numérico: 0 = A, 1 =B, 2 = C, ..., Z = 25
; Compilación:                      MASM E_DRIVE;

.MODEL LARGE                        ; Define el tipo de modelo de memoria
INCLUDE EXTASM.INC                  ; Archivo de cabecera proporcionado por
                                    ; Clipper

PUBLIC      E_DRIVE
E_DRIVE     PROC FAR                ; Define el tipo de dato a enviar

            PUSH  BP                ; Guarda el estado anterior
            MOV   BP,SP
            PUSH  DS
            PUSH  SI
            PUSH  DI

            XOR   AX,AX             ; Invoca funciones DOS
            MOV   AH,19H
            INT   21H
            XOR   AH,AH

            PUSH  AX                ; Retorna parámetros a Clipper
            CALL  __RETNI
            ADD   SP,2

            POP   DI                ; Reestablece registros
            POP   SI
            POP   DS
            POP   BP

            RET                     ; Valor de retorno a Clipper
E_DRIVE     ENDP                    ; Termina el procedimiento
            END                     ; Fin del programa
```

10.3.2 Función para convertir de minúsculas a mayúsculas.

Clipper proporciona una función para convertir una cadena de caracteres de minúsculas a mayúsculas: UPPER(). El inconveniente que tiene está función es que no puede transformar las letras que tienen acento. La solución que proponemos es crear una función en lenguaje ensamblador que realice dicha tarea.

La función E_Mayus() recibe como parámetro un dato tipo string y obviamente retorna un dato tipo string. La función transformará todas las letras a mayúsculas independientemente de que tengan acento o no. Para lo cual la función considera como un caso especial en la transformación a los caracteres: á, é, í, ó, y ú.

```
; Función :          E_MAYUS.ASM      Hecha en Ensamblador
; Creación:          12/04/94         Por:  Jorge Castaño
; Ultima revisión:   15/04/94    Por:  Fernando Tecuapacho
; Propósito:         Convierte una cadena de caracteres de minúsculas a
;                    mayúsculas. Recibiéndola desde un programa Clipper
; Compilación:       MASM E_MAYUS;

.MODEL LARGE                ; Define el tipo de modelo de memoria
EXTRN   __PARC:FAR          ; Define el tipo de dato a recibir
EXTRN   __RETC:FAR          ; Define el tipo de dato a enviar

PUBLIC E_MAYUS
.DATA                       ; Reserva espacio en memoria para trabajar
INBUFF  DB 256 DUP (?)

                            ; Comienza el procedimiento
.CODE
E_MAYUS   PROC FAR
          ASSUME cs:@CODE, ds:DGROUP, es:DGROUP
                            ; Guarda el estado anterior
          PUSH  BP
          MOV   BP,SP

          PUSH  DS
          PUSH  ES
          PUSH  SI
          PUSH  DI
```

```
                MOV   AX,@DATA
                MOV   DS,AX
                MOV   ES,AX

                MOV   AX,1              ; Inicializa
                PUSH  AX
                                        ; Recibe la cadena
                CALL  __PARC            ; DX:AX
                ADD   SP,2              ; Reset SP

                MOV   DS,DX             ; Direcciona registros
                MOV   SI,AX
                MOV   DI,OFFSET INBUFF
                                        ; Copia cadena a 'INBUFF
CICLO:          LODSB
                CMP   AL,0
                JE    FUERA
                STOSB
                JMP   CICLO
FUERA:          STOSB
                                        ; Inicializa direcciones
                MOV   AX,@DATA
                MOV   DS,AX
                MOV   ES,AX
                MOV   SI,OFFSET INBUFF
                MOV   DI,OFFSET INBUFF
                                        ; Realiza la conversión de minúscula
CAMBIO:         LODSB                   ; a mayúscula
                CMP   AL,0
                JE    SALIDA
                CMP   AX,0A0H           ; á
                JE    LETRA_A
                CMP   AX,082H           ; é
                JE    LETRA_E
                CMP   AX,0A1H           ; í
                JE    LETRA_I
                CMP   AX,0A2H           ; ó
                JE    LETRA_O
                CMP   AX,0A3H           ; ú
                JE    LETRA_U
                CMP   AL,20H            ; Caracter blanco
                JE    GUARDA
                AND   AL,0DFH           ; Otro caracter
```

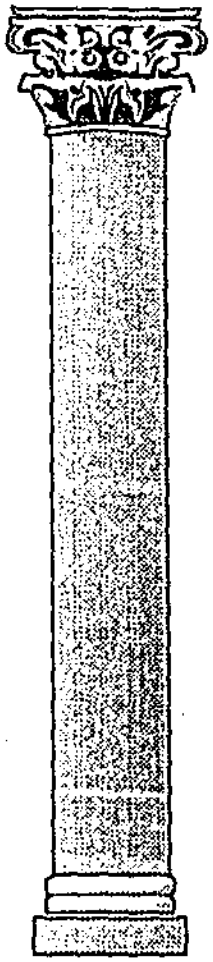
```
GUARDA:   STOSB

          JMP   CAMBIO

LETRA_A:  MOV  AX,041H          ; Cambia a 'A'
          STOSB
          JMP   CAMBIO
LETRA_E:  MOV  AX,045H          ; Cambia a 'E'
          STOSB
          JMP   CAMBIO
LETRA_I:  MOV  AX,049H          ; Cambia a 'I'
          STOSB
          JMP   CAMBIO
LETRA_O:  MOV  AX,04FH          ; Cambia a 'O'
          STOSB
          JMP   CAMBIO
LETRA_U:  MOV  AX,055H          ; Cambia a 'U'
          STOSB
          JMP   CAMBIO
                                ; Regresa la cadena
SALIDA:   PUSH DS
          MOV  AX,OFFSET INBUFF
          PUSH AX
          CALL __RETC           ; Envía el dato
          ADD  SP,4

          POP  DI               ; Restablece registros
          POP  SI
          POP  ES
          POP  DS
          POP  BP

          RET
E_MAYUS   ENDP                  ; Termina el procedimiento
          END                   ; Fin del programa
```

10.3.3 Programa para ejemplificar el uso de las funciones hechas en Ensamblador

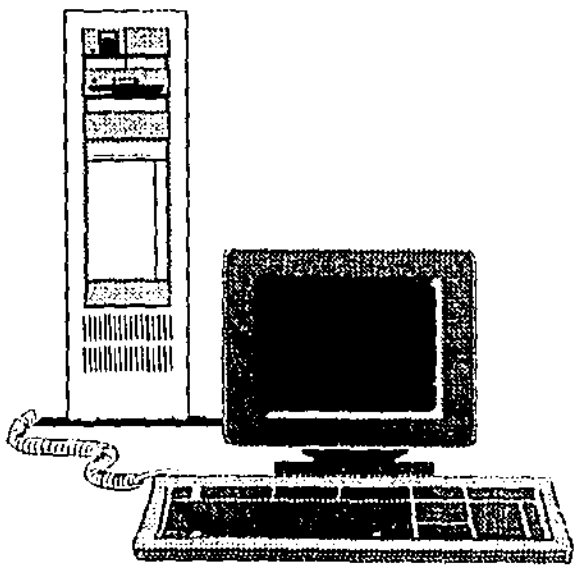
A continuación se muestra el listado de programa Clipper que comprueba las dos funciones hechas con Ensamblador.

```
/*
  Programa:          PRUEB102.PRG      Hecho en Clipper
  Creación:          18/04/94          Por:  Jorge Castaño
  Ultima revisión:   21/04/94          Por:  Fernando Tecuapacho

  Propósito:         Ejemplificar el uso de las funciones hechas con
                     Ensamblador:  E_Drive.asm  y E_Mayus.asm
  Compilar:          CLIPPER Prueb102 /N/M/W
  Encadenar:         RTLINK Prueb102,E_Drive,E_Mayus LIB CLIPPER,EXTEND
*/

FUNCTION PRUEB102()
LOCAL disco
CLS
disco :=CHR(E_drive() +ASC("A")) +":"
@5,18 SAY "La unidad áctual es " +disco   //  Error ortográfico intencional
//  Se retorna la misma cadena pero convertida a mayúsculas:
@7,18 SAY E_Mayus("La unidad áctual es " +disco)
RETURN NIL     /*      Fin de prueb102.PRG       */
```

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO



■ CONCLUSIONES



ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES
A R A G O N

Clipper es un lenguaje de alto nivel orientado al manejo de archivos. Es poco exigente en requerimientos de equipo. Puede ejecutarse un programa Clipper en una computadora con procesador 8086, con tan sólo 512 Kb de RAM y sin necesidad de tarjeta de video, aunque cabe aclarar que para el diseño del programa se recomienda que sea un equipo superior.

Es muy importante investigar que librerías existen en el mercado y que sean compatibles con Clipper para aprovecharlas como herramientas que pueden solucionar algunos de nuestros requerimientos, con lo cual podemos ahorrarnos tiempo que está relacionado con el costo del producto final.

Es recomendable crear una librería de funciones de usuario, de preferencia en lenguaje C o ensamblador, ya que se ejecutan más rápido y nos ahorran código de programación.

El encadenador RTLINK proporciona nuevas herramientas entre las cuales queremos hacer hincapié en el manejo de overlays y de librerías preencadenadas. El manejo de overlays nos permite optimizar el manejo de la memoria RAM. Mediante las librerías preencadenadas podemos reducir en un 50% ó más el tamaño del archivo ejecutable.

Clipper 5.01 incorporó nuevas formas de declaración de variables, así como nuevos operadores e introdujo un nuevo concepto que son los operadores compuestos. También introdujo nuevos tipos de datos: Nulo (NIL), objeto (OBJECT) y bloque de código (Code Block).

El preprocesador de Clipper cobra una gran importancia ya que nos permite integrar en nuestros archivos fuentes las siguientes técnicas: constantes simbólicas, comandos definidos por el usuario y archivos de cabecera, que le dan una mayor claridad a nuestros programas y al mismo tiempo se reduce código.

La versión 5.01 incorpora los arreglos de N dimensiones, lo cual era una limitante muy importante en las anteriores versiones de Clipper ya que solo se podían hacer arreglos de una dimensión.

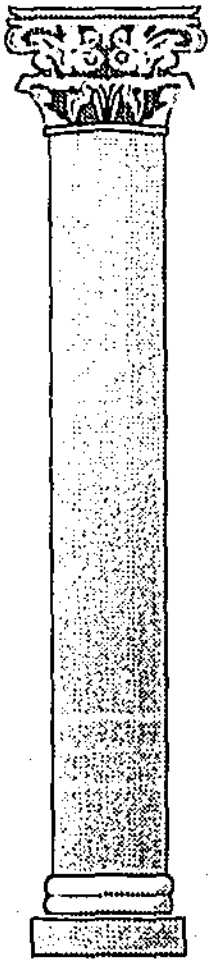
Clipper 5.01 incorpora cuatro clases: GET, ERROR, TBROWSE y TBCOLUMN, de las cuales únicamente analizamos a dos de ellas: TBROWSE y TBCOLUMN.

La clase TBROWSE contiene objetos que facilitan las operaciones de manipulación de archivos .DBF, al estilo de la función DBEDIT(), pero con muchas más posibilidades. Permite la presentación y edición de datos en forma de tablas, crede ación filtros y desplazamiento de columnas. La recuperación de datos se realiza mediante la evaluación de los CB. Cada objeto TBROWSE depende de los objetos creados por la clase TBCOLUMN.

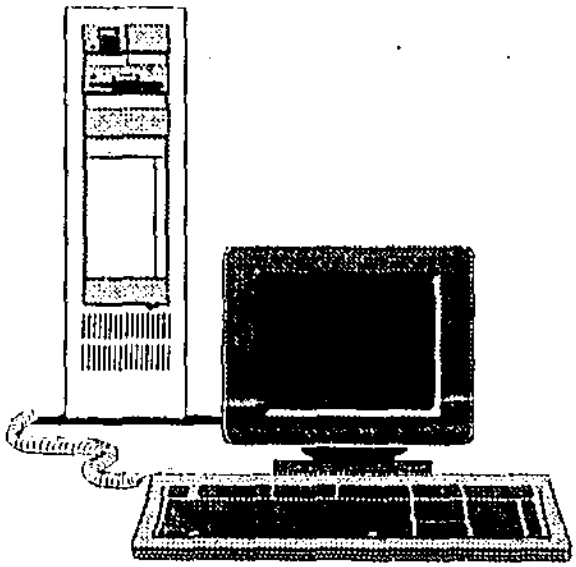
Contrariamente a lo que sucede con otros productos, sólo existe una versión de Clipper, tanto para los entornos monousuario, como para los de red de área local. En este es uno de los mejores productos del mercado. dBASE III +, dBASE IV y otros manejadores de bases de datos necesitan un LAN PACK para correr sobre red de área local.

Clipper corre directamente sobre el sistema operativo de red, sin necesidad de ningún producto adicional. Sin embargo en cuanto a la cuestión de bloqueo de archivos y registros, esta debe de considerarla siempre el programador, contrariamente a lo que sucede con otros programas similares.

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO



- BIBLIOGRAFIA



ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES
A R A G O N

1). CLIPPER 5.2 POWERTOOLS
STEPHEN J. STRALEY
BANTAM COMPUTER BOOKS
MARCH 1993.

2). USING CLIPPER 5.01
W. EDWARD TYLEY
QUE
1992.

3). CLIPPER 5.0 REFERENCIA RAPIDA
FRANCISCO MARIN, ANTONIO QUIROS CASADO
MACROBIT
1991.

4). CLIPPER 5.01 AVANZADO
JOSE RAMALHO
McGRAW-HILL
1992.

5). 111 FUNCIONES EN CLIPPER 5.01
JOSE RAMALHO
McGRAW-HILL
1992.

6). CLIPPER 5.0 METODOS Y UTILIDADES DE PROGRAMACION
JOSE JAVIER GARCIA
ADDISON-WESLEY
1992.

7). CLIPPER 5.01 REFERENCE
COPYRIGHT (c) NANTUCKET CORP.
1985-1991.

8). ARLIB V5.1 THE ARTISTIC LIBRARY
MANUAL DE REFERENCIA
COMPUTER SYSTEMS RESEARCH
1992.

9). CURSO DE PROGRAMACION CON C
MICROSOFT C
FRANCISCO JAVIER CEBALLOS
MACROBIT
1990.

10). REDES DE AREA LOCAL
LA SIGUIENTE GENERACION
THOMAS W. MADRON
MEGABYTE
1992.

11). MICROSOFT MACRO ASSEMBLER BIBLE
NABAJYOTI BARKATI AND RANDALL HYDE
SAMS
1992.

12). SEMINARIO CLIPPER 5.01 AVANZADO
COMPUTER SYSTEMS RESEARCH
1993.

13). SEMINARIO CLIPPER 5.01
PROGRAMACION ORIENTADA A OBJETOS
COMPUTER SYSTEMS RESEARCH
1993

14). REVISTA OBJECT: PROGRAMMER
No. 1, 2, 3, 4 y 5
COMPUTER SYSTEMS RESEARCH
1993-1994.

15). CLIPPER 5.01
JOSE ANTONIO RAMALHO
McGRAW-HILL
1992.