Expressions

In *Clipper*, all data items are identified by type for the purpose of forming expressions. The most basic data items (i.e., variables, constants, and functions) are assigned data types depending on how the item is created. For example, the data type of a field variable is determined by its database file structure, the data type of a constant value depends on how it is formed, and the data type of a function is defined by its return value.

These basic data items are the simplest expressions in the *Clipper* language. More complicated expressions are formed by combining the basic items with operators. This section defines all of the valid *Clipper* data types and

operators that you will use to form expressions, as well as any special rules that you will need to know.

Data Types

The data types supported in the *Clipper* language are:

- Array
- Character
- Code Block
- Numeric
- Date
- Logical
- Memo
- NIL

In *Clipper*, an array structure is as a separate data type, and there are distinct operations for arrays that are invalid for other data types. Code blocks are also a data type, but the operations that you can perform using them is limited. Arrays and Code Blocks are discussed in separate sections of this chapter.

This section discusses each of the simple data types, providing you with the following information: when to use the data type; how to form its constants, or literals; what limitations are in effect; and what operations are available.

Character

The character data type is used to identify data items that you want to manipulate as fixed length strings. The *Clipper* character set is defined as the entire printable ASCII character set (i.e., CHR(32) through CHR(126)), the extended ASCII graphics character set (i.e., CHR(128) through CHR(255)), and the null character, CHR(0). See the ASCII Character Chart appendix in this book for a complete table of ASCII codes.

Valid character strings are formed by combining zero or more characters in the defined character set with a maximum of 65,535 characters per string (i.e., 64K minus one character used as a null terminator).

Character string literals, or constants, are formed by enclosing a valid string of characters within a designated pair of delimiters. In the *Clipper* language, the following delimiter pairs are designated:

- two single quotes (e.g., 'one to open and one to close')
- two double quotes (e.g., "one to open and one to close")
- left and right square brackets (e.g., [left to open and right to close])

✣ Note

To express a null string, use only the delimiter pair with no intervening characters—including spaces. For example, "" and [ ] both represent a null string.

Since all of the designated delimiter characters are part of the valid character set, the delimiter characters themselves may be part of a character string. If you want to include any of these characters in a character string literal, you must use an alternate character for the delimiter. For example, the following string:

- I don't want to go.

could be expressed as:

- "I don't want to go."

Similarly, the string:

- She said, "I have no idea."

can be represented with:

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

This constraint should not be too limiting since you have three pairs of delimiters from which to choose.

The following tables give a synopsis of all operations in the *Clipper* language that are valid for the character data type. These operations act on one or more character expressions to produce a result. The data type of the result is not necessarily character.

✣ Note

The memo data type discussed later in this section is used to represent variable length character data. It is a true variable length data type that can only exist in the form of a database field. The character operations listed in this section may also be used with memo fields.

Table 1-1: Character Operations

Operation	Description
+	Concatenate
-	Concatenate without intervening spaces
=	Compare for equality
==	Compare for exact equality
!=,<>, #	Compare for inequality
<	Compare for sorts before
<=	Compare for sorts before or the same as
>	Compare for sorts after
>=	Compare for sorts after or the same as
= or STORE	Assign
:=	In-line assign
+=	In-line concatenate (+) and assign
-=	In-line concatenate (-) and assign
$	Test for substring existence
REPLACE	Replace field value
ALLTRIM()	Remove leading and trailing spaces
ASC()	Convert to numeric ASCII code equivalent
AT()	Locate substring position
CTOD()	Convert to date
DESCEND()	Convert to complemented form
EMPTY()	Test for null or blank
ISALPHA()	Test for initial letter
ISDIGIT()	Test for initial digit
ISLOWER()	Test for initial lower case letter
ISUPPER()	Test for initial upper case letter
LEFT()	Extract substring from the left
LEN()	Compute length
LOWER()	Convert letters to lower case
LTRIM()	Remove leading spaces
PADC()	Pad with leading and trailing spaces
PADL()	Pad with leading spaces
PADR()	Pad with trailing spaces
RAT()	Locate substring position starting from the right
REPLICATE()	Duplicate
RIGHT()	Extract substring from the right

Table 1-1: Character Operations (cont)

Operation	Description
RTRIM()	Remove trailing spaces
SOUNDEX()	Convert to soundex equivalent
SPACE()	Create a blank string
STRTRAN()	Search and replace substring
STUFF()	Replace substring
SUBSTR()	Extract substring
TRANSFORM()	Convert to formatted string
TYPE()	Evaluate data type using macro substitution
UPPER()	Convert letters to upper case
VAL()	Convert to numeric
VALTYPE()	Evaluate data type directly

Character strings are compared according to the ASCII collating sequence. Memo fields can also be compared to character strings. When EXACT is OFF two character strings are compared according to the following rules; assume two character strings A and B where the expression to test is (A = B):

- If B is null, return true (.T.).
- If LEN(B) is greater than LEN(A), return false (.F.).
- Compare all characters in B with A. If all characters in B equal A, return true (.T.); otherwise return false (.F.).

With EXACT ON, two strings must match exactly, except for trailing blanks.

Memo

The memo data type is used to represent variable length character data. It is a true variable length data type that can only exist in the form of a database field. A memo field takes up ten characters in the database file which is used as a pointer to the actual data which is stored in a separate (.dbt) file.

Besides the fact that the length may vary in a memo field from one record to another, memo fields are identical to character fields. The character set is identical, the 65,535 maximum character limitation stands, and comparisons are performed in the same way.

Since it can apply only to a field, there is no literal representation for the memo data type. However, character string literals can be assigned to memo fields with REPLACE.

The following table gives a synopsis of all operations in the *Clipper* language that are valid for the memo data type. The operations listed here are those designed to work specifically with memo fields, but they may also be used with long character strings.

Table 1-2: Memo Operations

Operation	Description
HARDCR()	Replace soft with hard carriage returns
MEMOEDIT()	Edit contents
MEMOLINE()	Extract a line of text
MEMOREAD()	Read from a disk file
MEMOTRAN()	Replace soft and hard carriage returns
MEMOWRIT()	Write to a disk file
MLCOUNT()	Count lines
MLPOS()	Compute position

In addition to these specialized memo operations, all of the character operations listed previously may be used with memo fields.

Date

The date data type is used to identify data items that represent calendar dates. In *Clipper*, you can manipulate dates in several ways, such as finding the number of days between two dates and determining what the date will be ten days from now. The date character set is defined as the digits from zero to nine and a separator character specified by SET DATE.

In *Clipper 5.0*, the SET EPOCH command has been added to the language to change the default century assumption. See the entry for this command in the *Standard Commands* section for more information.

Dates are formed by stringing together digits and separators in a particular order that depends on the current SET DATE value. By default, SET DATE is AMERICAN and dates are of the form *mm/dd/[cc]yy*: *mm* represents the month and must be a value between 1 and 12; *dd* represents the day which must fall between 1 and 31; *cc*, if specified, represents the century and must be between 0 and 29; *yy* represents the year which must fall between 0 and 99; and / is the separator.

Clipper supports all valid dates in the range 01/01/0100 to 12/31/2999 as well as a null, or blank, date. You may specify the century as part of a literal date regardless of the status of SET CENTURY; however, you will not be able to enter non-twentieth century dates in @...GET variables, nor will you be able to display the century unless SET CENTURY is ON.

Dates do not have a straightforward literal representation like character strings and numbers. Instead, you must use the CTOD( ) function to convert a character string constant expressed in date form into an actual date value. To

so that, you form the date according to the rules described above and enclose it in one of the character string delimiter pairs. Then, you use the character date as the CTOD() argument as illustrated in the examples below:

```
CTOD("1/15/89")

CTOD('03/5/63')

CTOD([09/28/1693])

CTOD("01/01/0100")
```

CTOD() checks its argument to be sure it is a valid date. For example, "11/31/89" and "02/29/90" would not be valid dates because November has only 30 days, and 1990 is not a leap year. CTOD() converts an invalid date to a null date.

✣ Note

To express a blank, or null date, use a null string as the argument for CTOD() (e.g., CTOD("")).

The following table gives a synopsis of all operations in the *Clipper* language that are valid for the date data type. These operations act on one or more date expressions to produce a result. The data type of the result is not necessarily date.

Table 1-3: Date Operations

Operation	Description
+	Add a number to a date
-	Subtract from a date
++	Increment
--	Decrement
= or ==	Compare for equality
!=, <>, or #	Compare for inequality
<	Compare for earlier
<=	Compare for earlier or the same as
>	Compare for later
>=	Compare for later or the same as
:= or STORE	Assign
:+	In-line assign
+=	In-line add and assign
-=	In-line subtract and assign

Table 1-3: Date Operations (cont.)

Operation	Description
REPLACE	Replace field value
CDOW()	Compute day of week name
CMONTH()	Compute month name
DAY()	Extract day number
DESCEND()	Convert to complemented form
DOW()	Compute day of week number
DTOC()	Convert to character in SET DATE format
DTOS()	Convert to character in sorting format
EMPTY()	Test for null
MONTH()	Extract month number
TRANSFORM()	Convert to formatted string
TYPE()	Evaluate data type using macro substitution
VALTYPE()	Evaluate data type directly
YEAR()	Extract entire year number, including century

Dates are compared according to their chronological order.

Numeric

The numeric data type is used to identify data items that you want to manipulate mathematically (e.g., perform addition, multiplication, and other mathematical functions). The *Clipper* numeric character set is defined as the digits from zero to nine, the period to represent a decimal point, and the plus and minus to indicate the sign of the number.

With the exception of fields, *Clipper 5.0* stores numeric values using the IEEE standard double precision floating point format, making the range of numbers from 10^-308 to 10^308. Numeric precision is guaranteed up to 16 significant digits, and formatting a numeric value for display is guaranteed up to a length of 32 (i.e., 30 digits, a sign, and a decimal point). This means that numbers longer than 32 bytes may be displayed as asterisks, and digits other than the 16 most significant ones are displayed as zeroes.

When a value is stored in a numeric field of a database (.dbf) file, it is converted from IEEE format to a displayable representation. When a numeric field is retrieved from a file, it is converted back to IEEE format before any operations are performed on it. Since the displayable format of a numeric value is guaranteed up to a length of 32 bytes, this is the largest recommended numeric field length.

Numeric literals are formed by stringing together one or more of the following: a single leading plus or minus sign, one or more digits to represent the whole portion of the number, a single decimal point, and one or more

digits to represent the fractional part of the number. Some valid numeric constants are shown below:

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

Note

Unlike character strings, literal numeric values are not delimited. If you enclose a number—or any other string of characters—in delimiters, it becomes a character string.

The following table gives a synopsis of all operations in the *Clipper* language that are valid for the numeric data type. These operations act on one or more numeric expressions to produce a result. The data type of the result is not necessarily numeric.

Table 1-4: Numeric Operations

Operation	Description
+	Add or Unary Positive
-	Subtract or Unary Negative
*	Multiply
/	Divide
%	Modulus
++	Increment
--	Decrement
= or ==	Compare for equality
!=, <>, or #	Compare for inequality
<	Compare for less than
<=	Compare for less than or equal
>	Compare for greater than
>=	Compare for greater than or equal
= or STORE	Assign
:=	In-line assign
+=	In-line add and assign
-=	In-line subtract and assign
*=	In-line multiply and assign
/=	In-line divide and assign
	In-line exponentiate and assign

Table 1-4: Numeric Operations (cont.)

Operation	Description
REPLACE	Replace field value
ABS()	Compute absolute value
CHR()	Convert to ASCII character equivalent
DESCEND()	Convert to complemented form
EMPTY()	Test for zero
EXP()	Exponentiate with e as the base
INT()	Convert to integer
LOG()	Compute natural logarithm
MAX()	Compute maximum
MIN()	Compute minumum
MOD()*	Compute dBASE III PLUS modulus
ROUND()	Round up or down
SQRT()	Compute square root
STR()	Convert to character
TRANSFORM()	Convert to formatted string
TYPE()	Evaluate data type using macro substitution
VALTYPE()	Evaluate data type directly

Numbers are compared according to their actual numeric value. Note that the = operator and the == operator behave identically when comparing numbers—they both check for equality to the maximum precision allowed by the IEEE 8-byte floating point format.

Numeric operations—including comparisons—are completely unaffected by any SET command. If numeric operations or comparisons appear to produce unexpected results, it is probably because of the automatic display formatting of floating point values. The display formatting is affected by the SET FIXED and SET DECIMALS commands.

Logical

The logical data type is used to identify data items that are boolean in nature. Typical logical data items are those with values of true or false, yes or no, or on or off. In *Clipper*, the logical character set consists of the letters y, Y, t, T, n, N, f, and F.

By definition, only two logical values are possible, but in *Clipper* there are several ways to represent these two values. To form a literal logical value, enclose one of the characters in the defined logical character set between two periods. The periods are delimiters for logical values just as quote marks are for character strings.

The literal values .y., .Y., .t., and .T. represent true. The literal values .n., .N., .f., and .F. represent false. The preferred literal representations are .T. and .F.

The following table gives a synopsis of all operations in the *Clipper* language that are valid for the logical data type. These operations act on one or more logical expressions to produce a result. The data type of the result is not necessarily logical.

Table 1-5: Logical Operations

Operation	*Description*
.AND.	And
.OR.	Or
.NOT. or !	Negate
= or ==	Compare for equality
!=, <>, or #	Compare for inequality

For the purpose of comparing logical values, false (.F.) is always less than true (.T.).

NIL

NIL is a new data type introduced in *Clipper 5.0* that allows you to manipulate uninitialized variables without generating a runtime error. It is a data type that has only one possible value, the NIL value.

The literal representation for NIL is the string of characters "NIL" without delimiters. When referenced by a display command or function, this is how NIL values display. Note also that NIL is a reserved word in *Clipper 5.0*.

The following table gives a synopsis of all operations in the *Clipper* language that are valid for the NIL data type. Except for these operations, attempting to operate on a NIL results in a runtime error. For example, you cannot REPLACE a field with NIL, nor can you concatenate NIL to a character string.

Table 1-6: NIL Operations

Operation	Description
= or ==	Compare for equality
!=, <>, or #	Compare for inequality
<	Compare for less than
<=	Compare for less than or equal
>	Compare for greater than
>=	Compare for greater than or equal
= or STORE	Assign to a non-field variable
:=	In-line assign to a non-field variable
EMPTY()	Test for NIL
TYPE()	Evaluate data type using macro substitution
VALTYPE()	Evaluate data type directly

For the purpose of comparison, NIL is the only value that is equal to NIL. All other values are greater than NIL.

In addition to the NIL operations, there are several declaration statements that automatically assign a NIL value to variables and array elements. These statements are summarized in the following table and discussed below.

Table 1-7: NIL Declaration Assignments

Operation	Description
DECLARE	Assign NIL to array elements
LOCAL	Assign NIL to local variables and array elements
PARAMETERS	Assign NIL to missing parameters
PRIVATE	Assign NIL to private variables and array elements
PUBLIC	Assign NIL to public array elements
STATIC	Assign NIL to static variables and array elements

All variables with the exception of PUBLIC and FIELD variables are initialized to NIL when declared or created without an initializer. PUBLIC variables are initialized to false (.F.) when created, and FIELD variables cannot contain NIL values at all. When an array is created with a declaration statement, including PUBLIC, all elements are initialized to NIL.

When a function or procedure is invoked, if an argument is omitted either by leaving an empty spot next to a comma or by leaving an argument off the end, a NIL value is passed for that argument. Note, however, that the function or procedure cannot distinguish between a NIL value that is passed explicitly

(e.g., an expression in the argument list that evaluates to NIL) and one that is passed as the result of omitting an argument.