Functions and Procedures

User-defined functions and procedures are the basic building blocks of *Clipper* programs. Every program is made of one more procedures and user-defined functions. These building blocks consist of a group of statements that perform a single task or action. They are similar to functions in C, Pascal, or other major languages.

The visibility of function and procedure names falls into two classes. Functions and procedures visible anywhere in a program are referred to as *public* and declared with FUNCTION or PROCEDURE statements. Functions and procedures that are visible only within the current program (.prg) file are referred to as *static* and declared with a STATIC FUNCTION or STATIC PROCEDURE statements. Static functions and procedures are said to have *file-wide* scope.

Static functions and procedures are quite useful for a number of reasons. First, they limit visibility of a function or procedure names thereby restricting access to the function or procedure. Because of this, subsystems defined within a single program (.prg) file can provide an access protocol with a series of public function or procedures, and conceal the implementation details of the subsystem within static functions and procedures. Second, since the static function or procedure references are resolved at compile-time, they preempt references to public function and procedures which are resolved at link-time. This assures that within a program file, a reference to a static function or procedure executes that routine if there is a name conflict with a public function.

Defining Functions and Procedures

Functions and procedure definitions quite similar with each having a slightly different requirements for return values.

Defining a User-Defined Function

User-defined function can be defined anywhere in a program (.prg) file, but definitions can not be nested. A function definition has the following basic form:

```
[STATIC] FUNCTION <identifier> [(<local parameter list>)]
   [<variable declarations>]
   .
   . <executable statements>
   .
   RETURN <return exp>
```

A function definition consists of a function declaration statement with an optional list of declared parameters. Following the function declaration are series of optional variable declaration statements such as LOCAL, STATIC, FIELD, or MEMVAR. As mentioned before a function must return a value and therefore a RETURN statement with a return values must be specified somewhere in the body of the function. A function definition begins with the FUNCTION declaration statement and ends with the next FUNCTION or PROCEDURE statement or end of file.

If the user-defined function declaration begins with the STATIC keyword, the function is visible only to procedures and user-defined functions declared within the same program (.prg) file.

The following is a typical example of a user-defined function definition:

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

Defining Procedures

Procedures are identical to user-defined functions with the exception that no return value is required and therefore not RETURN statement is required. Like user-defined functions, procedures can be defined anywhere in a program (.prg) file, but definitions cannot be nested. The following is the general form of a procedure definition:

```
[STATIC] PROCEDURE <identifier> [(<local parameter list>)]
   [<variable declarations>]
   .
   . <executable statements>
   .
   [RETURN]
```

Like a function definition a procedure definition begins with a PROCEDURE declaration statement and ends with the next FUNCTION or PROCEDURE statement, or end of file.

If the procedure declaration begins with the STATIC keyword, the procedure is visible only to procedures and user-defined functions declared within the same program (.prg) file.

Calling Functions and Procedures

Although functions and procedures are quite similar they can be called in different ways.

Calling a Function

User-defined functions you create and specify are not different from the supplied standard functions and therefore can be called in the same way. Functions can specified either in expressions or as statements. For example, the following are all legitimate function calls:

```
? "This is the " + Ordinal(DATE()) - "day"      // Expression
Report()                                        // Statement
result := Report("Quarterly")                   // Expression
```

When a function is called, it must always be specified including the open and close parentheses. If arguments are to be passed to called functions, they are specified between the parentheses and separated by commas. For example:

```
? MajorFunc("One", "Two", "Three")
```

Calling a Procedure

A procedure can be called either using function calling syntax by specifying the procedure call as a statement. Here you call the procedure as you would a *Clipper* user-defined function specified as a statement, like this:

```
<procedure>([<argument list>])
```

The second way is using the DO...WITH command. This method is not recommended since it passes arguments by reference as a default.

Passing Parameters

When you invoke a procedure or user-defined function, you can pass values and references to it. This facility allows you to create black-box routines that can operate on data without any direct knowledge of the calling routine, and vice-versa. The following discussion defines the various aspects of passing parameters in *Clipper*.

Arguments and Parameters

Passing data from a calling routine to an invoked routine involves two perspectives, one for the calling side and one for the receiving side. On the calling side, the values and references passed are referred to as *arguments* or actual parameters. For example, the following function call passes two arguments, a constant and a variable:

```
LOCAL nLineLength := 80
? Center("This is a string", nLineLength)
```

On the receiving side, the specified variables are referred to as *parameters* or formal parameters. For example, here the variables are specified to receive the string to center as well as the line length:

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

The specified receiving variables are placeholders for values and references obtained from the calling routine. When a procedure or user-defined function is called, the values and references specified as arguments of the routine's invocation are assigned to the corresponding receiving parameter in the invoked routine.

In *Clipper 5.0*, there are two ways to specify parameters depending on the storage class of the parameter you want to use. Parameters specified as a part of the procedure or user-defined function declaration are *declared parameters* and are the same as local variables. They are visible only within the called routine and have the same lifetime as the routine. Parameters specified as arguments of the PARAMETERS statement are created as private variables and hide any private or public variables or arrays of the same name inherited from higher level procedures and user-defined functions. In versions of *Clipper* prior to *Clipper 5.0* as well as other dialects, this was the only way to receive parameters.

✣ Note

In Clipper 5.0, you cannot mix declared parameters and a PARAMETERS statement within the same procedure or user-defined function definition. Attempting this generates a fatal compiler error.

In *Clipper*, parameters are received in the order passed and the number of arguments does not have to match the number of parameters specified. In fact, arguments may be skipped within the list of arguments when a routine is invoked. Receiving parameters without corresponding arguments are initialized to NIL. For example, the following function call skips two arguments from within the list of arguments. As you can see from the output, parameters not receiving a value or reference from an argument are given a NIL value:

```
? TestProc("Hello",,,"There")

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

When a routine is called, the PCOUNT( ) function is updated with the position of the last argument specified within the list of arguments. This includes skipped arguments, but not ones left off the end of the list. In the example above, PCOUNT( ) returns 4.

Passing by Value

Passing an argument by value means that the argument is evaluated and its value is copied to the receiving parameter. Changes to a receiving parameter are local to the called routine and lost when the routine terminates. In *Clipper*, all variables, expressions, and array elements are passed by value as a default if the function calling syntax is used. This includes variables containing references to arrays, objects, and code blocks.

As an example, the following passes a database field and a local variable to a procedure by value:

```
LOCAL nNumber := 10
USE Customer NEW
SayIt( Customer->Name )
? nNumber                                        // Result: 10
RETURN

PROCEDURE SayIt( fieldValue, nValue )
   ? fieldValue, ++nValue                        // Result: Smith 11
   RETURN
```

The importance of pass by value is that the called routine cannot change the caller's data by changing the value of the receiving parameter. This increases modularity by relegating the responsibility to the calling routine to determine whether it allows its data to be changed by a calling routine.

Passing by Reference

Passing an argument by reference means that a reference to the value of the argument is passed instead of a copy of the value. The receiving parameter then refers to the same location in memory as the argument. If the called routine changes the value of the receiving parameter, it also changes the argument passed from the calling routine.

Variables other than field variables can be passed by reference if prefaced by the pass-by-reference (@) operator and the function or procedure is called using the function-calling syntax. For example:

```
nX := nY := 1
Increment(@nX, nY)
? nX                                        // Result: 2

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

In this example, the change made to the *nNumber* parameter in the called routine is reflected in the *nX* argument after the Increment( ) procedure is called.

As you can see, passing arguments by reference can be quite dangerous if parameters are inadvertently assigned new values in the called routine. Because of this, passing by reference should only be used in special cases such as returning more than value from a procedure or user-defined function. A good example of this is the FREAD( ) function which takes a buffer variable passed by reference, fills the variable with characters read from a binary file, and returns the number of bytes read.

Note that arguments passed to routines called with the DO...WITH statement are passed by reference as a default. Note also that other dialects commonly pass arguments by reference as a default. In *Clipper*, this practice is strongly discouraged and therefore use of the DO statement is not recommended. All DO invocations should be replaced with function calling syntax and variables passed by reference prefaced with the pass-by-reference (@) operator.

Passing Arrays and Objects as Arguments

When using function calling syntax, variables containing references to arrays and objects are passed by value as are variables containing values of any other data type. This may seem confusing since it reveals that passing references involves a level of indirection. But when a variable containing a reference is passed to a routine, a copy of the reference is passed instead of the actual reference.

If, as an example, an array is passed to a routine by value and the called routine changes the value of one of the array elements, the change is reflected in the array after the return, since the change was made to the actual array via the copied reference to it. If, however, a reference to a new array is assigned to a variable passed by value, the new reference is discarded upon return to the caller and the array reference contained in the original variable is unaffected.

Conversely, if the variable containing a reference to an array or object is passed by reference by preceding the argument variable with the pass-by-reference (@) operator, any changes to the reference are reflected in the calling routine upon return. For example:

```
// Change the value of an array element
a := {1, 2, 3}
changeElement(a)
? a[1]                          // Result: 10

// Change an array; does not work
a := {1, 2, 3}
changeArray(a)
? a[1]                          // Result: 1

// Change an array by passing by reference
changeArray(@a)
? a[1]                          // Result: 4

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

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

Although, this behavior appears different from previous versions, it has always been the case. The confusion lies in the difference the way arrays are handled in *Clipper 5.0*. In previous versions of *Clipper*, you could not assign, return, or otherwise transport a reference to an array except by passing it to a procedure or user-defined function. Since this was the case, there was never any way to know how arrays were actually passed to subroutines. They always behaved as if passed by reference and so we believed. In *Clipper 5.0*, array references are treated the same as other data types, thereby revealing the actual and true rules of parameter passing.

Argument Checking

In *Clipper*, there is no argument checking, and therefore the number of parameters does not have to match the number of arguments passed. Arguments can be skipped or left off the end of the argument list. A parameter not receiving a value or reference is initialized to NIL. If arguments are specified, PCOUNT( ) returns the position of the last argument passed.

Since arguments can be skipped, PCOUNT( ) is not always an accurate gauge of what arguments have been specified. To ascertain this information, you can compare a parameter to NIL and if it is not equal you supply a default value or generate an argument error. The following example demonstrates this concept:

```
FUNCTION TestParams( param1, param2, param3 )
   IF param2 = NIL
     ? "Parameter was not passed"
     param2 := "default value"
   ENDIF
   .
   . <statements>
   .
   RETURN NIL
```

In addition to the NIL test, VALTYPE( ) can be used to test for a parameter receiving a value as well as the proper data type, like this:

```
FUNCTION TestParams( cParam1, nParam2 )
   IF VALTYPE(nParam2) != "N"
     ? "Parameter was not passed or invalid type"
   ENDIF
   .
   . <statements>
   .
   RETURN NIL
```

✣ Note

Remember to use VALTYPE( ) instead of TYPE( ) to test the data type of declared parameters. A TYPE( ) applied to a declared parameter will always return "U."

Passing Arguments from the DOS Command-line

Arguments can be passed directly from the DOS command-line to a program's main procedure. Parameters are specified either in a PARAMETERS statement or declared as a part on the main procedure's declaration. Remember, if the main procedure is declared, the main program (.prg) file must be compiled with the /N option.

Arguments passed are all received as character strings. Multiple arguments are specified with each argument separated by a space. If an argument contains an embedded space, it must be enclosed in quote marks in order to be passed as one string. For example, the following DOS command-line passes two arguments when invoking PROG.EXE:

```
C>PROG "CLIPPER COMPILER" 5
```

The main procedure receiving parameters from the DOS command-line should test the number of passed parameters with PCOUNT( ) to assure that critical parameters have been specified.

Returning Values From User-Defined Functions

As defined, a user-defined function must return a value. To do this, you must specify a RETURN statement with an argument. A RETURN statement

performs two actions: first, it terminates processing of the current routine transferring control back to the calling routine; and second, if the current routine is a user-defined function, it returns any value specified as the argument of the RETURN statement to the calling routine. The return value specified can be a constant or an expression. For example:

```
RETURN ("Today is " + DTOC(DATE()))
```

In *Clipper 5.0*, a user-defined function can return a value of any data type including arrays, objects, code blocks, and NIL. For example, the following user-defined function returns an array to a calling routine:

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

RETURN statements can occur anywhere in a user-defined function definition, allowing processing to be terminated before the function definition ends. For example, the following code fragment terminates processing if a condition is true (.T.) and continues if the condition is false (.F.):

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

Note that RETURN is limited by the fact that it can return only one value. More than one value can be returned if arguments are passed by reference, although this is not a preferred solution. An aggregate data structure can be defined as an array containing other arrays, and arrays, as mentioned above, can be passed throughout a program as a single value.

If the user-defined function's return value is not used, it is discarded. This is the reason you can specify a user-defined function as a statement.

Recursion

A procedure or user-defined function is recursive if it contains an invocation of itself. This can be either direct or indirect when a function calls another function that again calls the original function.

For example, the following example is a user-defined function that uses recursion to calculate the factorial of a specified number. A factorial is the product of all positive integers from one to a specified number. The factorial of 3 therefore is 1 * 2 * 3 which is 6.

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