# Preprocessor Primer

**Data Based Advisor**

July 01, 1992 | Gutierrez, Dan

Prior to the release of **Clipper** 5, the concept of an integrated **preprocessor** was foreign to most xBase developers. Preprocessors have, however, been around for a long time in the C world, and their usefulness continues to shine. A **preprocessor** does many things to improve application development; it promotes programs that are easier to read, maintain, and enhance. In addition, the **preprocessor** can speed up your applications.

A **preprocessor** is the part of a compiler that provides for such features as

*manifest constants*,
*include files* (also known as *header files*),
*conditional compilation,*
*pseudo functions,* and
*user-defined commands (UDCs).*

This article explains what a **preprocessor** is, what it can do for your programming, and how to get the most out of using it. Next time, I'll cover the portion of the **preprocessor** that lets you build UDCs.

**What's a preprocessor?**

Basically, a **preprocessor** is a "front end" to a compiler, reading the input source code, performing some manipulations, and then passing the resulting modified code to the compiler. Think of a **preprocessor** as a program that prepares your source code for the compiler. The result: The compiler needs to handle fewer language constructs. The **preprocessor** standardizes the code before the compiler sees it, making the compiler's job easier and faster.

Everything begins with the source code you've written in PRG files and any external **preprocessor** files, **Clipper** header files, and alternate command sets that supplement this source code. The **preprocessor** takes the source code and **preprocessor** files and produces a translation (that can be written to a disk file), which is then passed on to the actual compiler. This "preprocessed" version is actually a simpler version of your source code.

If you examine the preprocessed code, you'll find relatively few different commands. For example, there's no **?** command in the **Clipper** 5 language; all references to it are translated by the **preprocessor** to calls to the **QOUT()** console function. (Much of what **Clipper** 5 does is by way of calls to various internal functions. Nantucket has discouraged calling these functions directly, thus sidestepping the processor, because the internal functions may change from release to release; but the syntax of the **Clipper** 5 language won't.)

There's no way to avoid the **preprocessor**, but this is a necessary situation. Without the **preprocessor**'s actions, you'd never successfully compile a program. In order for you to see the work of the **preprocessor**, include the /P switch when you compile a program. This saves the preprocessed version of the program to a file with the same name as the program, except with a PPO extension.

When you view the contents of PPO files, you'll notice blank lines throughout. This is due to the **preprocessor**'s effort to preserve the line numbers so the correct line will be diagnosed in the event of a runtime error or while you use the debugger.

**Manifest constants**

The first ingredient of the **preprocessor** to examine is the manifest constant. Using manifest constants intelligently can enhance program readability and maintainability.

A manifest constant is a definition to the compiler that assigns a trying of text that may represent a constant, expression, part of a command, etc. to some identifier (similar to a memory variable). The difference between a manifest constant and a memory variable is that all manifest constants are resolved at "compile time," whereas memory variables may stay around for the duration of the program, i.e. during execution. "Resolved" here simply means that by the time the compiler gets hold of your source code, all the constant values you equated to **preprocessor** names have been substituted in place of the names. Again, this process simplifies the job of the compiler considerably, since unlike **memvar**s, constants take up no space in the symbol table maintained by **Clipper**. Only the actual constant values are passed to the compiler.

Another difference between constants and memvars is that constants are only known to the program module in which they're defined. If several modules need the constant, each module must include a definition for it. Since it can be tedious to include all such definitions in all the programs that need them, there's feature called the header file that addresses this problem.

Another benefit to using manifest constants is flexibility. If a constant is referenced in many program modules and that constant has to be changed, a memvar approach would require you to change the value in each and every program that needs it. Manifest constants, on the other hand, require only one change.

In order to define a manifest constant to the **preprocessor**, you use the **#DEFINE** directive. The general form of #DEFINE is:

```
#define []
```

Here, the directive name is #define and is usually written in lower case (although upper case is fine too) is the manifest constant name and is the optional text string with which the **preprocessor** will replace the name. For example, this:

```
#define MAXRECS 100
```

will replace all occurrences of MAXRECS with 100.

The job of the **preprocessor** in handling manifest constants is similar to a global search and replace function in a word processor. The **preprocessor** scans the source code and makes replacements for all names thus defined.

There's one other way to declare a manifest constant to a program. You can use the /D compiler switch to make a definition outside of the program. The syntax is:

```
/D [=]
```

Here, is the name of the constant you want to define. Notice that the part is optional. This is because the most frequent use of the /D switch is to define an empty value. This will be more clear when describing the **#IF#ELSE #ENDIF** directives.

One thing to remember when coming up with names for manifest constants is that all substitutions are ***case sensitive***. If a directive such as:

```
#define PageTitle "CUSTOMER HISTORY REPORT"
```

is defined, then the following occurrence in the source code wouldn't be replaced:

```
@ 1,40 SAY PAGETITLE
```

Manifest constants are the only place where **Clipper** 5 cares about case (case doesn't matter with memvars). The custom is to make manifest constants upper case.

There's no limit to the number of characters you can use in an identifier, and all will be significant (as opposed to a maximum of 10 significant characters for memvars).

To illustrate how you might use a manifest constant, consider the following code segment:

```
#define PAGELENGTH 60

IF nLineCnt == PAGELENGTH

   // End-of-page code goes here

   nLineCnt := 1

ENDIF
```

Prior to the **preprocessor**, the IF statement's logical expression could have been:

```
IF nLineCnt == 60
```

or, a memvar could have been defined called "nPageLength" containing the value 60. Aside from taking up often precious symbol table space, using a memvar "pseudo-constant" instead of a manifest constant tends to slow down your program slightly. The following code:

```
PRIVATE nConstant := 7 // PUBLIC yields same speed

LOCAL nI, nVal

FOR nI := 1 TO 10000
    nVal := nConstant
NEXT
```

will yield slower execution speeds than this code:

```
#define CONSTANT 7

LOCAL nI, nVal

FOR nI := 1 TO 10000
    nVal := CONSTANT
NEXT
```

The **preprocessor** automatically substitutes the references to CONSTANT and replaces them with a 7. In other words, nVal := 7 is the resulting code executed 10,000 times. In the former example, repeated references to the memory location represented by the memvar nConstant are required to supply the value to the assignment statement.

Another benefit to using a manifest constant over a memvar is that a memvar may be changed inadvertently during the execution of the program. A constant can never be altered unless another #define is embedded in the source code.

Also, a memvar's type can change dynamically during the operation of the program, making the program prone to the possibility of a runtime error in the future. As a manifest constant, the memvar's name, value, and type (implicitly speaking) remain the same at all times.

**Manifest constant examples**

One common use of manifest constants occurs while programming applications that interact with the keyboard. Consider the following code:

```
#define K_ESC 27
IF nLastKey == K_ESC
   CLOSE DATABASES
   RETURN NIL
ENDIF
```

The logical expression preprocess to nLastKey == 27. Examine the resulting PPO file for proof.

Constants are also good for defining standard error messages that may be referred to in various places in a program. If the following constant were defined in a program:

```
#define MSG_NOCUST "Error: customer not found"
```

```
it could appear in several commands, such as:
```

```
@ 10,25 SAY MSG_NOCUST
```

or:

```
@ 1,1 SAY "Severe " + MSG_NOCUST
```

You can also define a constant to contain part of a **Clipper** command. Below, we store a PICTURE clause in a manifest constant for substitution into an @..GET command. The reason this works is clear once you remember that the **preprocessor** treats the program like one big chunk of text.

```
#define ALLUPPER PICTURE '@!'

@ 10, 10 GET cCustName ALLUPPER
```

**The "un-identifier"**

There's one more **preprocessor** directive that pertains to manifest constants (and pseudo functions):

```
#undef
```

where is either a manifest constant name or pseudo function name. The purpose of this directive is to "un-define" an identifier. The reason for this is that **Clipper** doesn't allow the same identifier to be used more than once in a #define directive; a compiler warning results if the same identifier is redefined. To redefine an identifier, you must follow the original definition at some point with an #undef directive, followed by another #define. For example:

```
#define MAXRECS 50
#undef  MAXRECS
#define MAXRECS 100
```

**Include files**

Once a group of manifest constants (and pseudo functions) has been created for an application or part of an application, it's a good idea to place them together in a file instead of entering the #define directives in each program file. Once grouped, it's simple to reference them from a program module by using the #INCLUDE **preprocessor** directive. The general structure of this directive is:

```
#include ""
```

Here, can be any legal DOS file specification, such as a combination of drive letter, path name, and file name. Although not enforced by **Clipper**, it has become somewhat of a standard convention to use a CH file extension for **Clipper** 5 header files (often called include files).

**Clipper** 5 uses header files extensively. In fact, the complier comes with a collection of standard includes (see the list below) that define manifest constants for specific purposes. For example, INKEY.CH is one of the most widely used standard header files because it contains names for each of the keys on the keyboard. If you inspect INKEY.CH, you'll see that K_UP is defined as 5. This corresponds to the fact that an ASCII character code of 5 represents the keyboard's UP ARROW key. Each time you need to refer to this key (possibly after calling the LASTKEY() function), you can use the manifest constant instead of hard coding the numeric ASCII value in your program.

| Header file name | Constant prefix | Usage notes |
|---|---|---|
| ACHOICE.CH | AC_ | ACHOICE () UDF |
| BOX.CH | B_ | Box drawing |
| DBEDIT.CH | DE_ | DBEDIT () UDF |
| DBSTRUCT.CH | DBS_ | DBSTRUCT () |
| DIRECTRY.CH | F_ | DIRECTORY () |
| ERROR.CH | EG_ | Error codes |
| FILIO.CH | F_, FC_, FO_, FS | Direct file I/O functions |
| GETEXIT.CH | GE_ | get : exitState values |
| INKEY.CH | K_ | INKEY () return values |
| MEMOEDIT.CH | ME_ | MEMOEDIT () UDF |
| RESERVED.CH | | Naming conflicts |
| SET.CH | _ SET_ | SET () |
| SETCURS.CH | SC_ | SETCURSOR () |
| SIMPLEIO.CH | | Simplified I/O commands |
| STD.CH | | Standard command definitions |

You can also specify a header file for a compilation by using the /I compiler switch. Simply compile a program like this:

```
CLIPPER FOO /INETLIB.CH
```

This would make available all definitions found in a user-defined include file named NETLIB.CH available to the program being compiled.

Another technique is to use the special DOS environment variable INCLUDE, which directs **Clipper** to a hard disk directory that contains include files. From the DOS prompt (or more likely from a batch file) you'd enter:

```
SET INCLUDE c:/clipper5/include
```

One benefit to using include files is that a change to a #DEFINE must be made only once. Since all program files that depend on include files have an #include directive, all references will automatically be resolved. This does require, however, that the program be recompiled for the new constant values to take effect.

**Conditional compiling**

The next **preprocessor** capability, conditional compilation, provides a mechanism where certain statements or groups of statements can be selectively included or ignored in the resulting compiled form of the program.

Often you have to develop a program in a general way, to provide functionality for a varied set of applications, but for a specific application, there may be sections of code that you don't want to have compiled into the program. For example, let's say you've developed a piece of software that targets two industries, but not at the same time. Believe for the moment that you've written a generic General Ledger program for normal for-profit businesses, but have also included Fund Accounting provisions for non-profit organizations. You only want to maintain one set of source code, but depending on who you're selling the product to, the Fund Accounting portions may or may not be appropriate. Using the conditional compilation directives in **Clipper** 5, you can compile the General Ledger program with or without the Fund Accounting features with just a one line change.

Conditional compilation also gives you the ability to build "demonstration" versions of your software by compiling code into the application that checks, for example, for a maximum number of records in a database.

The directives in this area #IFDEF, #IFNDEF, #ELSE, and #ENDIF work like the standard xBase IF statement. The difference between the **preprocessor** directives and the IF statements is that the former are all performed at compile time as opposed to runtime. The syntax of these commands is:

```
#ifdef
[#else]
#endif
```

or alternately:

```
#ifndef
#[else]
#endif
```

where is a manifest constant name and are any valid **Clipper** 5 statements or commands. In the case of #IFDEF, the first set of is included in the compilation if is defined, whereas the second set is compiled if it isn't. The reverse is true when using #IFNDEF. It's here where the:

```
#define
```

form of the #define directive comes into play. Used this way, doesn't need a as described in the previous section dealing with manifest constants. Instead, the fact that it's being defined is the information needed later on for conditional compilation.

Think of the **preprocessor** as managing a series of "existence markers" for manifest constants defined this way. If a constant appears in the #DEFINE directive, then an "existence" or "true" marker is posted. For example, in both the #IFDEF and

#IFNDEF directives, must contain the name of a manifest constant that may or may not have appeared in a #DEFINE directive. For the "true" to be executed, the identifier must only be defined. Consider this example:

```
#define DEMOVERS

#ifdef DEMOVERS
   GO BOTTOM
   IF RECNO () > 50
     ? "This is a demo version. Too many records"
     QUIT
   ENDIF
#endif
```

Here, the **preprocessor** sees the definition of the DEMOVERS identifier. Notice that there's no value associated with it. Instead, the only piece of information needed is the fact that it's defined. Later, when the #IFDEF directive is encountered, the **preprocessor** checks to see if DEMOVERS is defined; if it is, the code that follows is compiled into the program. In this case, the code to check for a maximum record count is compiled; if the identifier hadn't been defined, this code would be omitted from your program.

**Pseudo functions**

The pseudo function capability of the **Clipper** 5 **preprocessor** provides a vehicle for defining functions that are resolved at compile time rather than when the program is running. Often, you can avoid using user-defined functions (UDFs) by implementing them as pseudo functions, which enhances performance. UDF calls can be expensive in terms of the added overhead required to issue a call to a function, pass parameters, and then return a value to the calling program. With pseudo functions, although a function is still called and parameters are still passed, the process occurs at compile time not runtime. My limited benchmark tests indicate that a UDF approach can be two and a half times slower than using pseudo functions.

The formal syntax of the pseudo function is:

```
#define () []
```

is the case-sensitive name with which you invoke the pseudo function. [] is the optional list of parameters that need to be passed to the function. Last, is the function, in other words, the expression that the function uses to determine a return value. Note that there shouldn't be a space after.

When a pseudo function is referenced in a program, textual substitution takes place. The **preprocessor** identifies a reference to a pseudo functions with an optional parameter list--whose parameter count must match with that of the function's definition--and substitutes the () for the []. In addition, each of the passed parameters are substituted for their counterparts occurring somewhere in.

Remember, all of this occurs at compile time, long before an EXE file is generated or code executed.

The parameters that appear in follow the same naming rules as for manifest constants. Moreover, you can't skip any parameters found in the definition when a pseudo function is called (which you can do with UDFs).

Sometimes pseudo functions are called "compiler macros." This term is truly a misnomer and should be avoided. The term "macro" has special connotations in the xBase world, normally indicating the process of performing textual substitution in the program using the & operator.

How do you use a pseudo function? Suppose you need to convert a numeric memory variable to a string and at the same time strip off all leading blanks in the resulting string. Normally, the STR() function pads out a converted numeric to a total of 10 characters, so we need a pseudo function, NTRIM(), to get rid of the padded blanks. The following is a definition for NTRIM() that will perform this task:

```
#define NTRIM(n) (LTRIM(STR(n)))
```

A possible application of NTRIM() is:

```
LOCAL nSalary := 25000

? NTRIM (nSalary) // No leading blanks
```

Here's another example, where a pseudo function calls the REPLICATE() function, passing to it the number of times to replicate the character:

```
#define SMOOTHLINE(n) REPLICATE (CHR(196),n)
```

This pseudo function accepts a numeric parameter and substitutes it for the replication factor in REPLICATE(). Using SMOOTHLINE() would only make sense if the calling program requires numerous calls to REPLICATE() with the same replication string.

**Other features**

The last **preprocessor** directive we'll look at is #ERROR. When #ERROR is encountered during the compilation of a **Clipper** program, a compile time error is generated and an optional error message is displayed. The primary use of this feature is to stop a compile when a crucial condition necessary for the successful completion of the compile doesn't exist. The command is specified as:

```
#error []
```

where is an optional, non-delimited (in other words, quotes aren't needed unless they're to be displayed with the text) character string that's displayed in the event that the #error directive is encountered. #error causes the compiler to generate error number 2074.

As an example, consider this code segment:

```
#ifdef GRAPHICS
   #error Graphics aren't currently supported
#endif
```

Here if the GRAPHICS manifest constant is defined, the developer wanted to produce a version of the software with graphics capabilities. Evidently, however, the code doesn't exist yet for such a feature, hence it makes no sense to go through with the compile.