Let's Look at Clipper 5.0's preprocessor

Data Based Advisor

March 01, 1990 | Spence, Rick

By now you're probably aware that Clipper 5.0 will come with a built-in preprocessor (after all, I mentioned it in my column last month). In this article, I'll look at what a preprocessor is, how I think 5.0's will operate, and what benefits it will offer.

What is a preprocessor?

Consider a preprocessor a program you run before a compiler. It reads the program source file as input and produces a "preprocessed" file as output. The output file is then fed into the compiler. The compiler doesn't know-or care-that the preprocessing ever took place. You can add preprocessor commands, or directives, to your source program. (If you don't use preprocessor directives, the output file will be identical to the input.)

By using the preprocessor, you can program in a 'superset" of the language. The preprocessor removes the directives from the file so the compiler never sees them (the compiler doesn't understand directives). The preprocessor processes the source file according to the directives, ignoring commands it doesn't understand. C programmers have long enjoyed the benefits of a preprocessor. The C preprocessor implements constants, conditional compilation, and compiler macros. Some Clipper programmers use the preprocessor I developed, PRE/DB, which mimics the C preprocessor.

As you'll see in the following sections, Nantucket included C preprocessor features and some Clipper-specific additions. 5.0 architecture and features PRE/DB is a stand-alone program. It reads the source (PRE) file and produces a PRG file. The PRG file is then fed into the compiler. With Clipper 5.0 the preprocessor is built into the compiler. It's automatically invoked when you compile a program. This architecture has advantages and disadvantages. It's transparent to the programmer, and it can be more closely integrated with the compiler for efficiency (there's no reason it has to do a separate first pass).

The main disadvantage is that if you don't use preprocessor directives, you're unnecessarily penalized. The preprocessor is still invoked. This can be time consuming. There are other reasons the preprocessor isn't separate though, as you'll see later in this article. Even though 5.0's preprocessor is built-in, a compiler switch lets you view the preprocessed output file. This lets you see exactly what it's doing to your code (useful when debugging your preprocessor commands). Here's a list of the features implemented by the Clipper preprocessor:

- Symbolic or "manifest" constants
- Include files
- Compiler macros
- User-defined commands (UDCs)
- Conditional compilation

Let's look at each.

Symbolic constants

A constant, by definition, is something that doesn't change. Examples in dBase are: 13.14, "Hit any key to continue", .T.

Most languages let you associate a name with a constant. This makes your program easier to read and maintain. For example, a common piece of code might look like this:

```
* Wait for Esc

lkey = inkey(0)

DO WHILE lkey != 27
     lkey = inkey(0)
ENDDO
```

We know the value of the Esc key is 27, so this is easy to read. But what about the value of the PgUp or Ctrl-F7 keys? You can't possibly remember all key values. An inexperienced programmer (or a programmer with a short memory) would have an even harder time.

With the current version of Clipper (and FoxBASE+ and dBase), you probably define memory variables for your commonly used constants. You assign them when the program starts up and test them instead of the constant values, as in:

```
ESC = 27
CR = 10
PAGE_UP = 18

lkey = inkey(0)

DO CASE
     CASE lkey = ESC
     CASE lkey = CR
     CASE lkey = PAGE_UP
ENDCASE
```

I think we all agree that this form is much more readable than:

```
lkey = inkey(0)

DO CASE
   CASE lkey = 27
   CASE lkey = 13
   CASE lkey = 18
ENDCASE
```

The problem with this approach is that these variables consume precious memory, and they can be changed! There's nothing to stop you from writing: `ESC = ESC + 1` In which case, a subsequent test, like: `IF lkey = ESC` would produce a difficult-to-detect logic error. Constants are also used to define limits. Let's say you define an array to store some frequently used values. When you define the array, you have to give it an upper boundary, as in:

```
PRIVATE values[15]
```

Subsequent processing code will then use that constant value:

```
FOR i = 1 TO 15
   process(values[i])
NEXT
```

The trouble is, this constant is used in more than one place. If you need to increase the size of the array, you have to locate every occurrence of the array size constant and change it. Again, you can use a memory variable to store the constant, as in:

```
MAX_VALUES = 15

PRIVATE values[MAX-VALUES]

FOR i = 1 TO MAX VALUES
   process(values[i])
NEXT
```

But you incur the same memory overhead. What we really want to do in both cases is to define a constant, but give it a name. You can do this with 5.0's preprocessor. You define a constant with the #define command like you'd do in C. You define the keys we used in the first example as:

```
#define ESC 27
#define CR 13
#define PAGE_UP 18
```

Now, whenever the preprocessor comes across one of these symbols in the input file, it replaces it with the constant value. So, using the following code as input:

```
#define ESC 27
#define CR 13
#define PAGE_UP 18

lkey = inkey(0)

DO CASE
   CASE lkey = ESC
   CASE lkey = CR
   CASE lkey = PAGE_UP
ENDCASE
```

The preprocessor generates:

```
lkey = inkey(0)

DO CASE
   CASE lkey = 27
   CASE lkey = 13
   CASE lkey = 18
ENDCASE
```

This is what goes to the compiler. The #define commands are gone. The constant names are replaced with their respective values. Just like you'd write it in FoxBASE+. Now I ask you, which is more readable?

Notice that there are blank lines where the #defines were. This is to maintain the line numbers between the input file and the output file. The compiler, runtime system, and debugger all work with the preprocessed file. The programmer reads the original source. That's why it's important that the line numbers be the same.

There are many ways to use symbolic constants in dBase/Clipper programs. Key values are classic examples, such as:

```
#define BLACK N
#define BLUE B
#define GREEN G
#define CYAN BG
#define RED R
.                                                                                .
SET COLOR TO RED/BLUE
```

and:

```
#define VENDOR_ORDER 1
#define STORE_ORDER 2
#define DATE_ORDER 3
USE trans
SET INDEX TO vendors, stores, dates
. .
SET ORDER TO STORE_ORDER && instead of SET ORDER TO 2
```

You'll probably find many others as you experiment.

In contrast to the C preprocessor, 5.0's is not case sensitive. You can write:

```
#define Esc 27
```

and use it as:

```
IF lkey = Esc
```

The convention is to define all preprocessor symbols in upper case. I'll follow that convention.

Include files

It's important to note that constants, unlike memory variables, only affect the source file that defines them. Their value is only known to the preprocessor when the translation takes place. So, if you: `#define ESC 27` in one source file, then write: `IF lkey = ESC` in another, ESC will not be replaced with 27, and you'll get a runtime error. The compiler will think ESC is a variable, but the runtime system won't find it.

Remember, symbolic constants are only known at preprocessor time-not at runtime. Once you start using a preprocessor, your list of constants will grow to be quite large. Just to define all the frequently used key values requires 50 or so. It would be a real pain if you had to include these definitions in every source file that used them. Fortunately, you don't. You can place them in a separate file, known as a "*header file*," and include that file in any program that needs it. The preprocessor command to do this is #include. You give it the name of the header file. Then the preprocessor reads it as if those lines were included in the program.

Let's look at an example using the keys we defined earlier in this article. We'll place the following directives:

```
#define ESC 27
#define CR 13
#define PAGE_UP 18
```

in a file called "keys.h." To use these constants in a program file, you simply issue the directive:

```
#include "keys.h"
```

The constants then become available to the program, as if they were defined in the source file. The #include line is stripped from the output file so the compiler doesn't see it. When you include a file, the preprocessor first looks for it in the current directory. If it's not there, the preprocessor looks in any directories specified by the INCLUDE environment variable. You can specify many subdirectories here-just separate them with semicolons. This allows you to keep all your include files in one place.

You don't need a separate copy in each directory. You can also include program code in an include file, not just preprocessor commands. This isn't practical with PRE/DB your line numbers get out of sync. However, Nantucket has done a really good job here. Although the actual lines in the output file are at a different number than the input file, all errors, as well as the debugger, use the source file's line numbers. This is possible because the preprocessor emits line number information to the output file-telling the compiler to re-adjust its line numbers. This is exactly what C preprocessors do. Nantucket should be commended for the practice.

Here's one important thing to remember when using header files: The object file depends on both the program source file and header files. Your make response file should reflect this. If test.obj depends on test.prg, and test.prg includes two header files, say "keys.h" and "system.h," your make response file should include the following lines:

```
test.obj: test.prg keys.h system.h
clipper test
```

Then, if you change either of the two header files, the source file will be recompiled.

Compiler macros

You can view a compiler macro as a procedure placed directly into the program source code. You can parameterize a sequence of commands, just like a function, but the difference is that preprocessor macros are replaced directly into the source file. They aren't called as a subroutine the way a function is.

As a simple example, you can write a "min" macro as:

```
#define min(a, b) iif(a < b, a, b)
```

When you subsequently use or invoke the macro, as in:

```
x = min (y, z)
```

the preprocessor replaces it with the preprocessed right side of the macro. In this example, it replaces the line with:

```
x = iif(y < z, y, z)
```

You define the macro in formal parameters. When the preprocessor expands it, it replaces formal parameters with actual parameters. This is what happens when you call a procedure or function. The difference is that compiler macro substitution is done during preprocessing, not at runtime.

In 5.0 you can place multiple commands on a line, separating them with a semicolon. This lets you write compiler macros that use more than one command. As an example, you can write a macro to swap two variables as:

```
#define swap(a, b, c) c = b; b = a; a = c
```

This expands to three commands-all on the same line.

Compiler macros are handy for short sequences of code repeated in several places. They execute faster than equivalent functions, and you gain the advantage of parameterized sequences of code. Just like symbolic constants, compiler macros are only valid in the program that defines them. You can, of course, place them in header files, alongside the constants.

Don't confuse compiler macros, as implemented by the preprocessor, with the macro feature supported by dBase and Clipper. Compiler macros are handled by the preprocessor as textual substitutions. The compiler and runtime system don't even know they exist. dBase-style macros (&), on the other hand, are handled by the runtime system.

Conditional compilation

Conditional compilation lets you include or exclude code based on some condition. The usual purpose is to maintain different versions of a program without having separate copies of the source code-for example, when you have a complete version of a program and a demo version. Another example would be when you have a version of a program for dBase, as well as a Clipper-specific version.

The preprocessor commands to implement conditional compilation are very similar to the usual `IF ... ELSE ... ENDIF` statements. Code is included in, or excluded from, the output file based on the result of the test-like a filter. Certain code is passed through to the compiler; other code isn't. This way, you can generate different EXE files based on the values you test.

The important thing to note is that this testing is done by the preprocessor, not while the program is running. This means that the unselected code doesn't exist in the output file. Whereas the regular `IF ... ELSE ... ENDIF` statements test for a true or false result, the preprocessor commands check to see if a value is defined.

You define a value with the #define directive, as we've discussed. The preprocessor commands are:

* #ifdef-Test to see if a value is defined
* #ifndef-Test to see if a value is not defined
* #else-The negative side of the preceding
* #ifdef/#ifndef
* #endif-To match the preceding #ifdef/#else

You can define a symbol without giving it a value. For example, you can write:

```
#define DEMO_VERSION
```

then test it with:

```
#ifdef DEMO_VERSION
```

You're just testing to see if it's defined; you don't care what its value is. You only use this feature with conditional compilation. In other cases, you want the symbol to have a value!

Let's look at what the preprocessor does with a simple example.

Consider this code:

```
#define TEST-VERSION
#ifdef TEST VERSION
   WAIT "Program Starting - Hit Alt-D to enter the debugger"
#endif
main-menu()
```

Since TEST VERSION is defined, the code between the #ifdef and the #endif is passed through. The preprocessed file looks like this:

```
__ Wait( "Program Starting - Hit Alt-D to enter the debugger" )
main-menu()
```

The preprocessor passes the code between `#ifdef` and `#endif` and replaces its directives with blank lines. If `TEST_VERSION` isn't defined, the output file would look like:

```
main_menu()
```

The `WAIT` isn't passed through.

To create a demo version of a program, you could code the main loop as:

```
#define DEMO_VERSION
#define EXIT_CHOICE 1
#ifdef DEMO_VERSION
```

```
   #define NUM_LOOPS 10
   counter = 1
   choice = 1
   DO WHILE counter <= NUM_LOOPS .AND. ;
                  choice != EXIT_CHOICE
         choice = main_ menu()
         IF choice != EXIT_CHOICE
             process_choice(choice)
         ENDIF
         counter = counter + 1
    ENDDO
#else
    choice = 1
    DO WHILE choice != EXIT_CHOICE
         choice = main-menu()
         IF choice != EXIT_CHOICE
             process_choice(choice)
          ENDIF
    ENDDO
#endif
```

To create the live version, just remove the `#define DEMO_VERSION` from the source and recompile it. Another thing you can do is define a symbol on the compiler command line. This way, you don't need to modify the source code. You do it with the /d option, as in:

```
clipper test /dDEMO_VERSION
```

DEMO_VERSION is then defined in the source file.

User-defined commands (UDCs)

The `#command` directive is specific to Clipper. (It's not supported by the C preprocessor.) Look at it as a more general version of a compiler macro-its pattern recognition is much more advanced. A compiler macro has a name, like a function, and its parameters are passed in parentheses, also like a function. The `#command` directive, on the other hand, lets you define a "command" as a series of keywords, with the "parameters" in any order. For example, you can define a SWAP command as:

```
#command SWAP WITH USING => ;
= ;;
= ;;
=
```

Then you can write:

```
SWAP x WITH y USING temp
```

and the preprocessor will convert it to:

```
temp = y ; Y = x ; X = TEMP
```

As I said last month, this is how the SQL interface will be implemented-through a series of UDCs. Here's what's really interesting though: The entire Clipper language is now implemented this way. Every command you write is defined as a UDC. They're translated by the preprocessor into function calls. The commands are defined by the header file "std.ch". This implements the standard commands as we know them, but you're free to change them!

Unless instructed otherwise, the preprocessor "loads" these commands before processing the file (which can take a couple of seconds, even on a fast machine). You can use a compiler/preprocessor switch to tell it to take the definitions from another header file, so you can define your own source language!

The syntax and replacement rules for UDCs are complex (I don't have the documentation yet). We'll discuss them in far greater detail in a future article.

Summary

This month we looked at Clipper 5.0's potential preprocessor. It looks like Nantucket will implement the same features found in the C preprocessor. They'll also add the `#command` directive, which is how UDCs (and SQL interface commands) are implemented. Clipper programmers will enjoy the benefits of symbolic (manifest) constants, compiler macros, and conditional compilation. I'll use preprocessor directives in all my future articles.

Rick Spence was a member of the Nantucket development team for three years, and is author of PRE/DB, the database language preprocessor. He is currently working in Los Angeles on a variety of database topics and offers advanced Clipper training. His book on Clipper has been published by Data Based Advisor. Rick also writes a monthly column for Reference Clippery. He can be reached at Software Design Consultants, (818) 892-3398, or on MCI MAIL, user name LSPENCE.