MAINTAINABLE CODE I: Naming Conventions

Lincoln Stoller, Ph.D

4/9/93

Overview:

This article analyzes the problem of writing clear 4th Dimension code and suggests how naming conventions can lead to clearer code. It is the first of four articles on writing maintainable code.

Maintainable code is code that can be modified with minimum effort. In order to accomplish this your code needs two things:

- a structure understood by the person making the modifications;
- a structure that's consistent with present and future specifications.

This article focuses on naming schemes for procedures and variables, and discusses problems in the use of global variables. Writing code that reflects a program's logical structure is the topic of the second article. Writing code that is consistent with present and future specifications will be the topic of the third and fourth articles in this four-part series.

Naming

The importance of naming should be appreciated before you start programming. Once you've added a few hundred (or a few thousand!) variables without any naming conventions and you're having trouble figuring out what each one does, it is a little late for the insight that much of your confusion could have been avoided. The problem is that a careful naming convention supports and reflects the whole structure of your code: to add it after the fact requires untangling a host of elements and their relations.

What does naming entail? Naming is not capricious or euphonic (or boring), like people's names. Naming is not arbitrary or incidental like the variables $a^2 + b^2 = c^2$ in the Pathagorean theorem. (Remember what these refer to?). Naming the elements in a program is your best opportunity to write code that documents itself. By choosing names carefully your code can tell the computer how to act and communicate to the reader what its doing, both at the same time.

Variable and procedure names

Give your variables and procedures names that reflect their function. It's easier to follow code employing the string variables str30Name and str20Phone than variables named Var1 and Var2. The same holds true for procedures. This is exemplified by the code in Figure 5.

Without explanatory names	With explanatory names
Var1 := Procedure25(Var1)	str30Name := CapFirstLetters(str30Name)
Var2 := Procedure63(1;Var3)	str20Phone := ApplyPhoneFormt("USA";str20Phone)
Procedure128(4; »var62; 6)	AsignShpCod2Itm(4; »txtItmName; "Fed-X")

Figure 5. Code without and with explanatory names.

4D imposes a limit of 11 characters for global variable names and 15 for procedure names. There are some artful conventions that cram as much description as possible into these 11 or 15 characters. First, it's usually possible to omit vowels, except the leading ones, and retain a string that looks and sounds close enough to be read as the original word. We can certainly leave out silent letters and one member of a double consonant pair when it occurs. Blank spaces between words can be omitted using the convention of capitalizing the first character of each of the reduced words. You can define a glossary of abbreviations for commonly used words. Replacing, for example, the words "verify," "initialize," "record," "restore" and "enter" with the abbreviations "Vrf," "Int," "Rcd," "Rst" and "Ent." Employing these devices can cut the number of characters in half, as shown in in Figure 6. I've tried to reduce the length of the strings as much as possible. You may prefer to add a few vowels at the expense of a longer, more readable string.

Original String	Vowels and spaces removed	Abbreviations used
Initialize Contact Record	IntlzCntctRcrd	IntCntctRcd
Restore Values From Disk	RstrVlsFrmDsk	RstVlsFrmDsk
Verify Correct Values Entered	VrfyCrctVlsEntrd	VrfCrctVlsEnt

```
Case of      `Layout procedure for [Contact]Input
 : (Before )
   IntCntctRcd

 : (During )
   Case of
    :   (bnaRestore=1)
        RstVlsFrmDsk

    : (bnaEnter=1)
      If (VrfCrctVlsEnt)
        ACCEPT
     End if
   End case
End case
```

Figure 6. Examples of condensed procedure names and their use in context.

A word of warning: after you remove letters, different words will look the same; the further you reduce, the more things look the same. In order to avoid unintelligible or ambiguous contractions, you should adopt as complete a set of standards as possible before you begin naming. Remember to leave room for growth: if you think two-character codes will be adequate, use three.

Global variables

Global variables have two propertie: what they are as determined by their type, and what they do as determined by their function. Your variable names can indicate type, function or both. Knowing a global variable's type is useful when debugging, but it isn't as important as knowing what the variable is supposed to do. In fact, I've found that about the only time it's useful to incorporate variable type as part of the name is when the variable is constantly reused, and its only stable feature is its type. The only other time I include a type identifier is in cases where context does not fully reveal it, such as for some array types, pointers and the lengths of fixed-string variables.

The problem of keeping track of variable types has been made easier by 4D-XRef, or 4D Insider for Version 3, and the compiler. XRef/Insider can show you actual variable usage, allowing its type to be identified by context. The compiler will catch most illegal assignments. The exception is that Version 1 of the compiler doesn't alert you when a longer string is assigned to a shorter string, even though performing such an assignment will generate a runtime error. Keeping track of fixed-length string variables remains the developer's responsibility.

A possible set of type-defining variable prefixes is shown in Figure 7. In this system the first characters of a variable name tell what the variable is --that is, its type-- leaving the rest of the name to describe what it does. Type identifiers are not particularly useful since type usually follows from function or context. However, there are some exceptions.

Text and string variables are used for similar purposes but are not interchangeable when the number of characters exceeds the character length of one of the variables. Because arrays can be used in either of two ways, either directly, or by specifying one of their elements, they should be given a distinct prefix. Pointers and the values they reference can also be confused, so you should assign pointers a type-identifying name. Finally, if you define temporary-use global variables, those that carry different values depending on circumstances, then the names of these general-use globals should reflect their type.

Global Variable Type	Type prefix
Text	t
Real	r
Arrays	a + letter for type
Boolean	o
Picture	p
Integer	i
Longint	l
Time	m
Date	d
String of length xx, where xx={1,...80}	sxx
Pointer	n
External area	e

Figure 7. Character prefixes to identify global variable types

Buttons are associated with a longint global variable that is assigned the value 1 when the button is pressed. There are a variety of classes and subclasses of buttons, as shown in Fig. 8.

Button class (all are longint)	Button class prefix
button	bb
radio button	letter for group + br
check box	bc
pop-up menu	a (array) + letter for type
or scrollable area	
invisible button	bi
highlight button	bh
radio picture	rp

Button subclass (all are longint)	Button prefix
no action	Button class + na
accept	Button class + a
cancel	Button class + c
next record	Button class + nr
previous record	Button class + pr
first record	Button class + fr
last record	Button class + lr
delete record	Button class + dr
next page	Button class + np
previous page	Button class + pp
first page	Button class + fp
last page	Button class + lp
open included	Button class + oi
delete included	Button class + di
add to included	Button class + ai

Figure 8. Character prefixes for button classes and subclasses.

The classes of regular, invisible and highlight buttons all allow a variety of automatic actions. These buttons would be assigned both a class and a subclass prefix. Check box buttons do not have a subtype and would be assigned a name beginning with bc. Radio buttons also have no subtype, but 4D distinguishes groups of radio buttons by the first character of their variable name. This convention is essential for 4D to handle the marking and unmarking of buttons in a group. For radio and radio picture buttons we apply naming prefixes to the second and third letters and leave the first letter free for defining the radio group. Examples are shown in Fig. 9.

Class	Subclass	Purpose	Name
button	no action	compute sum	bbnaCmptSum
radio button	-	print details	arbPrtDtail
popup menu	-	select journal	atSelJournl
scrollable area	-	choose shipping carrier	atSelShiper
invisible button	delete included	delete related project	bidiDelProj
check box	-	print line number	bcPrtLinNms

Figure 9. Examples of naming conventions with explanation.

Type prefixes are particularly useful in button names since buttons tend to be reused, as will be discussed later. The conventions proposed here are trivial—almost obvious—and easily memorized. They're generated using the rule that the first character or characters of the object's name constitutes its prefix. If these characters are used by another type of object, then the second or third characters are used instead. This scheme can be applied to other objects like graphs and external areas.

These conventions only help if they're applied consistently. If you use these rules inconsistently there will be times where you won't know whether the leading characters are a prefix or a contraction. Since we don't have a tool for global search and replace that applies to the whole database structure, you are pretty much stuck with whatever variable names you implement at the start of your project. You must adopt naming conventions well before you actually need them. Perhaps you should give these codes a second glance!

SIDE BAR:
Will a large number of global variables degrade performance?
The number of global variables you employ does not appear to have an appreciable effect on the speed of your program, despite what we have occasionally heard over the years. I created two compiled databases whose only difference was that the first database contained 1K of global variables, while the second contained 18K (the maximum is 32K). The sole difference was an extra 1,500 real variables in the second database. I ran a FOR loop that recalled and assigned 5 real values 1,000,000 times and which took the same amount of time to run in both cases to 0.5%, which was the accuracy of my measurement. I conclude that these amounts of global variables do not affect performance.

The problems of global variables

Standard wisdom suggests that global variables are a menace and should be replaced with local variables and passed parameters. The case against global variables stems from the lack of any visible link connecting where globals are defined with the places they're used. As a result it's easy to loose track of what globals are being used, how they're assigned, what values they're assigned and the circumstances of their use. I call this the "cross-reference" problem.

Another problem with global variables that we don't often hear about concerns their reuse in different sections of the code for different purposes. The problem is how to know when a global variable is already being used for another purpose in order not to overwrite its contents with a new value. I'll call this the "reuse" problem.

As an example of the overwriting problem, suppose I assign a No Action button in a layout in the [Customer] file displayed through a DIALOG command that I name vbna1_dCu. Suppose that while this layout is displayed I open a second dialog that also employs this variable. When the second layout opens, it reinitializes the button value to zero. When I return to the original layout, the original instance of the variable inherits the properties and value last assigned in the second dialog. If the button were disabled in the second dialog box, it would appear disabled when we returned back to the first dialog; if the button were pressed to exit the second dialog box, the original instance would now have the value 1.

4D's implementation of global variables is unique in the following respects:

- global variables must be used in a number of common situations, including button and object variables and computed values on input or output layouts;
- global variables must be used as parameters in a number of built-in calls;
- only global variables can be "passed by reference." A variable can be passed by reference to another procedure when that procedure can change the variable's value. This is done in 4D by passing a global variable pointer to the called procedure.

Contrast this with the limited utility of 4D's local variables:

- local variable values cannot be changed outside the procedure that defines them (because they can't be passed by reference);
- local values do not persist through different layout procedure phases;
- locals can't be used for arrays or sets (although local arrays are now supported in 4D Version 3);
- locals can't be used on layouts or included layouts.

In summary, global variables are a necessary fact of life in 4D. In the next section I'll review strategies for minimizing the problems inherent in their use.

Living with global variables in 4D: the cross-reference problem

4D's strongest defense against the problems of global variables is 4D X-Ref for Version 2, and 4D Insider for Version 3. These tools analyze structure files and list the locations and contexts of global variables and other objects. X-Ref/Insider essentially resolves the cross-reference problem and is an indispensable part of the 4D programming environment. However X-Ref/Insider cannot solve all problems, and its utility is diminished when you reuse global variables. This is because reusing globals means that they are being redefined and reused in necessarily unrelated places, and that these locations will be mixed together in the X-Ref/Insider listings.

There are a number of conventions that can help you manage your globals. These include grouping global variable definitions and initializations in special procedures (preferably a separate procedure for each module) using names that clarify variable context and, when finished with variables, setting them to a special value indicating they're no longer in use. For example, each of my modules has its own initialization procedure that runs at start-up. These procedures define variable types, fill arrays, sets and variable values. My input layouts have initialization procedures that run in the Before phase to initialize the global used in the layout and layout procedure.

In some cases I've represented global variables using elements in an array. The array is created dynamically, intialized, used or modified as needed and then erased. This is a useful technique when you have to pass a lot of information with a minimum of communication between procedures. However, using arrays adds the new problem of remembering which values are located in which elements. Globals are still needed to display or modify the values (though you can display arrays in popup lists and scrolling areas).

Living with global variables in 4D: the overwriting problem

Your use of global variables is limited by an immutable 32K limit: compiled 4D will not allow more than 32K bytes of global variables. In 4D Version 3 this becomes 32K for variables defined in each process, and 32K for interprocess variables. This memory is static, is allocated when the program starts and cannot be expanded. Figure 10 lists the memory taken by variables of different types. From this list you figure out that 32K will allow for 8,000 longints, or 3,200 reals or 800 40-character string variables.

If you're wondering why text, picture and array variables occupy so little memory when they can contain so much data, it is because the data are stored in dynamic memory, memory taken from the application heap as available. The static memory that these variables require is fixed, and it's used to hold a handle to the location in dynamic memory where the data are stored. (See sidebar for a definition of handles.)

Variable type	Bytes of memory occupied
Arrays	10 + size of contents, cf. 4D Language
Reference p.13-3.	
Boolean	2
Date	6
Integer	4
Longint	4
Picture	4 + size of picture
Pointer	4
Real	10
String	smallest even number greater than string length + 4 bytes to store the length
Text	6 + number of characters
Time	4

Figure 10: Global variable types and memory requirements in a compiled database. These sizes do not include 2 bytes for space in the global variable address table.

A large program can exhaust its 32K of static memory if a new global variable is defined for every function. Consequently, we have to reuse global variables as a matter of course. Since you want to give a reusable variable a name that is consistent with its role, you need to make a decision about the context to which you'll limit its appearance. You can choose names that have complete generality such as v1real, v2real and string30, but this may not be the best choice as we'll see next.

SIDE BAR:

Pointers and Handles

Pointers and handles are basic concepts in the Mac operating system and can be implemented in 4D. Both pointers and handles are variables whose values are memory locations, and both are variables of type “pointer.” The difference between them is that data are found at the address referred to by a pointer, while a pointer is found at the address referred to by a handle. This means that if you dereference a pointer (which means you retrieve what’s at the location pointed to) then you get some datum, like a date or a real number. When you dereference a handle, the value you get back is a pointer, and this pointer needs to be dereferenced a second time to get the final datum value.

In the example on the left I assign a date to the variable MyBirthDay, assign MyPointer the location of this data variable and finally dereference the pointer to assign the date to a local variable. In the example on the right I introduce a handle which I then have to double dereference to recover the date.

```
MyBirthDay:=!1/4/56!
MyPointer:=>>MyBirthDay
$BirthDay:=MyPointer>>
```

```
MyBirthDay:=!1/4/56!
MyPointer:=>>MyBirthDay
MyHandle:=>>MyPointer
$BirthDay:=(MyHandle>>)>>
```

4D doesn’t offer different tools for pointers and handles, so the distinction between them is purely one of context. However, the Mac operating system, which you can deal with directly when programming in C or Pascal, makes extensive use of pointers and handles for memory management, accords them quite different roles and offers the programmer special tools for each.

Returning to the case of text and picture variables, when 4D needs to assign a value to a text or picture variable it first tells the operating system how much space it needs. The operating system reserves this amount of memory, assigns the address of this memory to a pointer, and then returns to 4D the location in memory where this pointer can be found. In other words, the operating system returns a pointer to a pointer which is, by definition, a handle.

You have two strategies for reusing global variables: you can either opt for completely general reuse, or accord the variables some degree of specificity that

should be reflected in their name. For example, you can limit variables to specific files, modules, or limit their use to specific programmers (in multi-programmer projects): Cust1real, Str30Inven or JJboolean1. Localized names are useful for buttons that are frequently reused in different screens. In this case naming conventions afford a measure of protection against the overwriting of variable values by similarly-named variables in multiple windows.

I distinguish button variables by button type, layout type (input, output and dialog), and file. I would name a series of 10 no-action buttons for [Customers] output layouts bbna1_oCn to bbna10_oCn. This clarifies the purpose of the variable and the surrounding code, and provides the flexibility to reuse the variable in a manner consistent with its name.

Characters	Indication
bbna	regular button, no-action
1,2, ... n	elements in an n-element series
_o	associated with an output layout
Cn	associated with the [Customer] file

Figure 11: Significance of the characters in a sample global variable name.

Limiting the scope of global variables through naming conventions is currently the best way to manage the overwriting problem. By employing a naming convention that distinguishes global variables used in different files and layouts, for example, we can localize the problem of overwriting to layouts of the same of file. Similarly, if global variables used in different modules always have different names, then they will never overwrite each other.

If you have experienced overwriting problems, then you might be dissatisfied with this solution. It is not a complete solution because no matter how you modularize or segment your code, there is always the possibility that reusable variables will overwrite each other with their own module. Add to this the possibility of having procedures in one module call procedures in another module. When this occurs, variables associated with different modules are being used by both. You cannot use X-Ref or Insider to solve the overwriting problem since it does not track the actual assignment of global variables. The only way to completely eliminate the

overwriting problem is either to avoid reusing global variables or to completely isolate procedures that reuse global variables. In theory you are best off avoiding globals whenever possible. In fact, you'll find that complex code will require their use by the hundreds.

In the future 4D will probably support some degree of what's called static scoping in order to provide a middle ground between the current implementation of global and local variables. Static scoping means that a variable that's defined in one procedure can be used just like a regular global variable in all procedures nested within the defining procedure, but is completely inaccessible to procedures outside the calling chain. Each procedure creates its own "world" of global variables that remain accessible to all other procedures until the defining procedure exits, after which these variables are removed from memory. A simple form of dynamic scoping would be to allow pointers to local variables. This would allow us to pass these pointers to subordinate procedures which could then modify the value assigned to the local variable. The author of 4D has said that such a scheme could be implemented within the current structure of 4th Dimension.

Acknowledgments

Thanks to the following folks for comments and suggestions: Scott Ribe, Tracy Harms, Michael Billesback and Jim Goshorn. Special thanks to Jennifer Fox for editing this "boring computer stuff."

References

Kernighan, Brian and P. J. Plauger, 1978. The Elements of Programming Style, 2nd ed., New York: McGraw-Hill.

Ledgard, Henry, 1987. Professional Software Vol I: Software Engineering Concepts, Reading, MA: Addison-Wesley.

Ledgard, Henry, 1987. Professional Software Vol II: Programming Practice, Reading, MA: Addison-Wesley.