C Programming 101

or Some Things You Need To Know About C Programming

Why Do I Have To Deal With This?

Well, actually, you don't really. Now that VEE 7 gives you access to the .NET library you don't need to deal with any of this stuff. And you can use all kinds of ActiveX thingies without ever having to know what sizeof(int) is. If you want to use the Compiled Function Interface though, you will inevitably be thrown into the C and there are a few simple things you can learn to keep yourself from drowning.

Data Size

Without a doubt, the most important consideration for the CFI is data size. If you call a function that expects a four byte parameter and you pass it two bytes, chances are something bad will happen. The CFI works at the machine level, bypassing all data integrity and appropriateness checks made by VEE and other languages. If something is not right it almost always spells disaster, but sometimes it doesn't and all that happens is you get the wrong answer. In some ways that's worse than having your program crash – you don't necessarily know anything at all went wrong. We'll see a VEE to C data size and type table later on. I should point out that if you're calling a __stdcall function (more about that later) VEE will catch many bad things that can happen when passing incompatible data size parameters when the function returns to VEE. It will say that the stack pointer has been corrupted (because it will have been). Sometimes this doesn't happen though. If you pass 2 and 4 byte parameters to a function that's expecting 4 and 2 byte parameters, the function still messes up but the stack pointer will not be corrupted if the function returns to VEE. And very often that's a big "if" the function returns to VEE.

Data Type

Other than data value (which we'll get to shortly) and data size, *data type* is sometimes very important. Some numbers are in specific formats that have to be interpreted correctly. Most often these take the form of floating point numbers. C understands two types of floating point numbers: float and double. Their sizes are 4 and 8 bytes, respectively. VEE calls these types Real32 and Real64 respectively. When you're passing these types across the CFI, their types must match or they will be interpreted incorrectly. For example, you can pass an array of Int32 to a compiled function that expects an array of float because float and Int32 are both the same size, but the function will try to interpret the integers as if they were IEEE 32-bit floating point numbers and whatever operation it performs on the array will probably be completely wrong.

Record Type

VEE gives you the Record Type for creating records of data that are to be considered as a unit, like an address book record for instance. It's members might include first name, middle name, last name, address, phone number and email address. All of these pieces of data belong to the same person, so it's convenient to keep them together. You can create an array of these records to make an address book. C allows the same thing, but VEE's Record Type variables are not allowed to cross the CFI so it would seem that you can not pass records across the CFI. Fortunately this isn't exactly so, but it's also not straightforward. Many, many Windows API functions deal with *C Structures* (exactly equivalent to VEE Record Type vars) and these types must be dealt with. Records *can* be passed across the CFI, but they cannot be in VEE Record Type format. We'll see how to accomplish passing Record Type or "struct" vars across the CFI, but there are a few other points to consider first, namely: data value vs. data location.

What's A Pointer?

There is no aspect of C more misunderstood than pointers. And I personally really have a hard time accounting for that. A piece of data has a *value* that is stored at an *address*. A pointer is a variable that holds the address of a data value, hence it *points* to the data. That's all there is to it. A pointer can point to anything, even another pointer. The regression can be endless, though you'll soon go stark raving mad if you have to deal with any more than two or three levels. In the Windows API, two levels rarely happens and I don't know of any examples of three levels. It's just too confusing. *Dereferencing* a pointer is getting the value of the data that the pointer points too, and this operation can be deceptively simple looking. Here again, data size is what counts. If you want to get the data value that a pointer points to, you have to know the size of the data that you're getting in addition to where it is. This is obvious when dealing with arrays: an array is always just a pointer. At the address of that pointer is a contiguous list of data values of equal type. If you want to dereference the tenth element of the array, you must add the start of the array to nine times the size of the data in bytes to find the tenth element.

Nine? What? Shouldn't that be ten? No! Arrays always start at 0 so the first element is number 0. The tenth element is number 9. That's a very confusing bit that nags everybody and it's called OBO (Off By One). Watch carefully for this in your programming endeavors. It will most definitely get you in one form or another. If you're going to be manually dereferencing arrays you will most definitely have to deal with it.

VEE and C both implement arrays in the same way, and that is a god send. It means that a VEE array is exactly the same as a C array and both languages deal with them in exactly the same way with almost exactly the same syntax. You can pass a VEE array over the CFI simply by naming it. The receiving function must usually be told how many elements are in the array, but that is a minor inconvenience. VEE has the totSize function to tell you how large the array is, and you can use a Get Values object to find out everything there is to know about an array, but that's only because when the array exists within VEE it is described by a VDC - a *VEE Data Container*. This container is not passed across the CFI and the called function has no way of knowing how many elements are in the array.

There are two other thing I want to mention about pointers. First, the size of a pointer. It *is* something of a complex issue, but for our purposes (VEE Programming for

Windows) you can assume that a pointer is always 32-bits. Secondly, I want to stress again that a pointer can point to anything - even a C struct or VEE Record Type var and *that* is how we ship VEE Records and C Structures across the CFI - as pointers that point to the data value. In the case of a VEE Record or a C struct, the "data value" is a block of memory that contains the data in the record or struct. This pointer can even be an array of these records or structs. It makes no difference. All that's going across the CFI is a 32-bit pointer.

The Type Table

Ok, that's a lot to take in all at once. Before we get involved in looking at some funky Windows types, lets see just how all this type stuff works out between VEE and C. The CFI isn't really picky about type, but it is *very* picky about size. **Any parameter passed to or returned from any compiled function must use types whose sizes match.** You might call this the Golden Rule of the Compiled Function Interface. The types don't necessarily need to match, but the size of the type *must match.* Sometimes you have no choice but to pick a type whose size fits and just use it regardless of what the type actually is.

Def Type is the type you express in a definition file. VEE Version is the minimum VEE Version on which the type can be used over the CFI. ***THIS IS NOT YET AN AUTHORITIVE SOURCE.*** When I get my hands on a v9 Advanced Programming Manual it will be. So on with the table...

VEE Type	C Type	Def Type	Data Size	VEE Version	Comments
VEE_BOOL	?		?	8	See note 3.
UInt8	unsigned char	byte	1 Byte	8	Thankfully, now usable as "byte".
Int16	short	short	2 Bytes	6	
UInt16	unsigned short	WORD	2 Bytes	8.5	Specifically, 2 unsigned bytes is a WORD.
Int32	long	long	4 Bytes	<=5	
Int64	long long	int64	8 Bytes	8	See note 4.
?	unsigned long	?	4 Bytes		Specifically, 4 unsigned bytes is a DWORD.
Real32	float	float	4 Bytes	<=5	
Real64	double	double	8 Bytes	6	
Text	char*	char*	4 Bytes	<=5	See note 1.
(none)	int	int	4	<=5	See note 2, 2a.

			Bytes?		

NOTE 1: The size of a text variable is of course the size of the string contained in the text variable. When VEE passes a text variable over the CFI, it is *always* passed as char* (here, the asterisk '*' is called a 'splat'. Kudos if you know why - you must have read my Type Spoofing article on the VRF). The size of what's being passed over the CFI on the other hand is that of a pointer, which is always the sizeof(int) and in a 32-bit world that means 4 bytes.

NOTE 2: Though the CFI allows you to define parameters as "int", there is no explicit VEE type that "int" directly translates to. The reason is that C defines the "int" type as an integer of the width of the basic registers of the machine. On a 32-bit machine, a C "int" is the same as C "long" or VEE "Int32". The advantage of using "int" instead of "long" in CFI definitions is that VEE will always (I hope) realize what the "sizeof(int)" is and since this is the same size as a pointer, parameters defined as "int" in CFI definitions are useful for passing explicit pointers (as opposed to implicit pointers - a VEE array is an implicit pointer). What all this gobbldy gook means is that if and when VEE becomes a native 64-bit application then explicit pointers typed as "int" will still work where as those typed as "long" will be only half as large as they should be.

NOTE 2a: We recently got a hint that VEE will be going 64-bit, so this distinction is now of paramount importance. Always define any pointer values as "int", not "long".

NOTE 3: I don't have any information on this type yet, but I imagine it's probably 1 byte. Recently, we got confirmation that VEE is now compiled with Microsoft Visual C++, and that compiler's "boolean" entity is 1 byte long. This is assuming a bit much though, since Windows has long defined BOOL as basically "int" and hence 4 bytes (in a 32-bit world). This ambiguity will be resolved as soon as possible. In the event that this really is "boolean", that's a C++ type and there isn't a C name for the same concept, but "unsigned char" will do.

NOTE 4: The type, __int64 is not a C type but an MS type. Usually, this is also referred to as "long int", "long long", "long64", "int64" or (as far as Windows was concerned until recently), LARGE_INTEGER or LONGLONG. This also gets into the difference between C and C++. C has a standard 64-bit integer and C++ does not. Here is another peculiarity of C: functions in the standard library which either are not defined in the "standard" or operate in non-standard ways are usually prepended with a single underscore (like int _kbhit(void) for instance in the Microsoft C RunTime Library [aka CRT]). Any entity which is prepended by two underscores is unique to that particular compiler (like __int64). Sometimes the differences are huge and very important, and sometimes they're trivial and completely irrelevant (like stdcall, _stdcall and __stdcall - they all mean the same thing but they're all different identifiers).

So these then are the basic types allowed by the CFI. In addition, you can "splat" (place an asterisk after) any particular type to indicate that the parameter will be a pointer to the named type. In particular, this is how you tell the CFI that you want to pass a VEE array to a compiled function, or pass a scalar by reference (except a string scalar - strings are always passed by reference). If you have an array of VEE Real32 values that you need to pass to a data acquisition dll, then define the parameter as "float*".

One curious case is "char**". This is how you pass a VEE Text array. If you think about this carefully, you can figure it out: a Text var is a char* as far as C is concerned. This is a value that points to a series of characters whose end marker is the character value zero. Since an array is a pointer to a block of memory that contains contiguous values of the same type, you know that a char** is a value that points to a block of memory and each data value is a 4-byte pointer to a series of characters whose end marker is the character value zero. It's tempting to think that the character data is stored immediately after the individual pointers in the list, but that violates the rule for arrays: in that case, there's no telling how large each data value is until they are all scanned for the zero byte. In actuality, it looks like this:

Text array (char**) $\longrightarrow$ Text[0] (char*) $\longrightarrow$ Text[0] (char block) \0x00

Text[1] (char*) $\longrightarrow$ Text[1] (char block) \0x00

Text[2] (char*) $\longrightarrow$ Text[2] (char block) \0x00

Text[3] (char*) $\longrightarrow$ Text[3] (char block) \0x00

What About Multidimensional Arrays?

Glad you asked. Both VEE and C do multidimensional arrays, but with both languages it's an illusion. And they both handle them in exactly the same way too - another god send. Basically you multiply all the maximum indices together multiplied by the size of the type to find the size of the memory block used to store the array. To dereference any particular element, you multiply the indices of the previous dimensions (from left to right) by the size of the type and then add the current index multiplied by the size of the type. Add that to the starting address of the array and you have the address of the element you're looking for. The number of indices doesn't matter - any array is passed as a pointer to the type.

Pointers And 64-bit Processors

As we move into the realm of a 64-bit native VEE, traversing arrays of pointers becomes a bit foggy. When you're trying to access successive pointers, you need to know if you have to add 4 or 8 to your index value to get the next one. If you are running a 64-bit version of VEE on a 64-bit processor, then you have 8-byte pointers. If you're using a 32-bit version of VEE on a 64-bit processor, then you'll be operating in a virtual 32-bit support subsystem and your pointers will be 4-bytes. Someday I'll address this with a simple SizeofPointer function, but until then just be aware that this little detail can trip you up.

Funky Windows Types

When you look at Windows application source code, or even look around in the function definitions of the Windows SDK, you see lots of words that describe types that are not listed in the table. This seems distressing until you know that this is yet another technique for *self documentation*. Self documentation is a big deal in textual languages. The idea is that the program itself should be it's own best documentation. We are quite used to that with VEE, but with a text based language it takes a lot of effort. Most programmers spend a lot of time developing the discipline necessary to follow self documentation. In general, the more you do it the better your programs become. Name decoration is a form of self documentation. So is *typedef*ing and that's what you see for a lot of Windows types.

In C, there are two ways to call a type by a different name. One of them is a simple text replacement technique called a *simple macro* which offers no type-checking capability or definition advantage and the other is called a typedef which offers both. C is very loosely typed, which means that you can define a variable as one type and use it as any other type as you wish. This offers great flexibility, but it also offers great danger. Type checking means that the compiler will check to make sure that you are actually doing what you think you are doing. But anyway...

A typedef is a statement that defines an alias for an intrinsic type. It can also define an alias for a user defined type, but that's not important for the types shown in the table. There is a file in the Windows SDK that contains a lot of the funky types you see in the function definitions, and it's called WinDef.h. There are far too many to list here, but trust me when I say that most of them boil down to 32-bit numbers. There are a great many unsigned types, and VEE does not do unsigned types (except of course UInt8 - which we can't use anyway). This is largely immaterial. These are not numbers that are meant to be interpreted by humans so you can usually ignore the difference. Here is a partial list of the most important types:

Name	**Type**	**Size**	**Comments**
DWORD	unsigned long	4 Bytes	Four bytes is called a Double Word.
INT	int	4 Bytes	Just an alias for int.
UINT	unsigned int	4 Bytes	
LONG	long	4 Bytes	Just an alias for long.
BOOL	int	4 Bytes	Boolean, Yes or No, True or False.
BYTE	unsigned char	1 Byte	
WORD	unsigned short	2 Bytes	Two bytes is called a Word.
FLOAT	float	4 Bytes	Just an alias for float.

LPFLOAT	float*	4 Bytes	Pointer to FLOAT.
LPBOOL	int*	4 Bytes	Pointer to BOOL.
LPBYTE	unsigned char*	4 Bytes	Pointer to BYTE.
LPINT	int*	4 Bytes	Pointer to INT.
LPWORD	unsigned short*	4 Bytes	Pointer to WORD.
LPLONG	long*	4 Bytes	Pointer to LONG.
LPDWORD	unsigned long*	4 Bytes	Pointer to DWORD.
LPVOID	void*	4 Bytes	Pointer to any type (see next section)..
UINT_PTR	unsigned int	4 Bytes	Used for "polymorphic types".
ULONG_PTR	unsigned long	4 Bytes	Used for "polymorphic types".
WPARAM	unsigned int	4 Bytes	Used to be "word param", used in messages.
LPARAM	long	4 Bytes	"Long param", used in messages.
LRESULT	long	4 Bytes	"Long result".
HANDLE	void*	4 Bytes	"Handle" to object.
LPTSTR	char*	4 Bytes	Pointer to string, i.e. VEE Text var.

These are just a few of the most frequently seen types, and I'm sure I've left a lot out. As you can see, most are simply 32-bit values. Whenever you come across a type you're not familiar with (eventually, you'll know the basic types by heart) you have to go look it up in the SDK or your header files to see what type it actually is.

A Word About Void

No doubt you've noticed that there are a couple "void*" types up there. In C, a void pointer is a pointer to any type. VEE does not accept "void*" in CFI definitions, so if you have to use a void pointer you have to call it something else. Since it's a pointer, it makes sense to call it an "int". If and when VEE changes to a 64-bit application, the sizeof(int) will follow the change and the void pointer will still work. VEE is not the only language that has trouble with void pointers. The C compiler itself will not allow you to dereference a void pointer. The reason is simple: to dereference a pointer, you need to know not only the location of the data but the size of the data, and since void pointers point to any type it is impossible to know the size of the data. When you use a void pointer, you have to know in advance what kind of data the pointer is pointing to and you have to treat it appropriately by telling the compiler that this pointer is to be considered a certain type. That's called a *type cast*.

Calling Convention

Ok, before we move on to *type spoofing*, I have to clear up something I said earlier about stdcall. This is known as a function calling convention. There are four of them used in C/C++: cdecl, stdcall, thiscall and fastcall. All are used to call functions, but normally you only see stdcall. That's because if you happen to catch wind of the calling convention of most Windows functions, you'll find that they are WINAPI, APIENTRY or CALLBACK. All of these are just aliases for __stdcall - the stdcall calling convention. All of these calling conventions have their own ins and outs, but since we're really only concerned with stdcall that's the only one I'll comment on.

The CPU maintains an area of memory called the *stack*. It's like extra scratchpad memory, beyond the actual registers in the machine. One of the things it's used for is calling functions. First, the function parameters are pushed on the stack one by one, then the current instruction pointer (the location of the instruction that the CPU is currently executing - this is held in a CPU register) is pushed on the stack and execution is transferred to the function address. Now, exactly *how* all this is done is called the calling convention (I guess it really ought to be called a protocol, but oh well). Also, there are two different ways to specify the parameters: by value or by reference (no doubt VB people are thinking ByVal and ByRef - that's exactly the right idea, though a little different - here we're considering the way the machine decides to push parameters on the stack, not the language). The stdcall calling convention always calls by value, but if the value you're pushing on the stack happens to be a pointer to some variable then you have effectively called by reference. Ponder that - it's important.

The stdcall calling convention dictates that:

1. Parameters are pushed on the stack from right to left as they occur in parentheses.
2. Function parameters are passed by value (unless a pointer or reference is explicitly passed).
3. An underscore character '_' is pre-pended to the internal name of the function.
4. An at sign '@' is appended to the internal name of the function.
5. The number of bytes passed as parameters are appended as an ASCII string to the internal name of the function.
6. The callee (the called function) cleans up the stack (gets rid of the parameters).
7. The function's return value is located in the primary arithmatic (EAX) register upon return.

Ok, let's take those one at a time. Parameters are pushed from right to left. Opposite of the way English reads. That means that the last parameter pushed is the first spelled out. Since the stack is used in reverse order, that means that the first parameter spelled out is in the lowest memory address. I know it's insane, just remember that the first parameter is first in memory. Sometimes that's important. Next we know that parameters are passed by value. That is, the value of the variable named is what VEE pushes on the stack. In the case of an array, VEE pushes the address of the array - thus passing the array by reference. Ponder that - it's important. An underscore, at sign and the number of bytes passed as parameters are added to the function name. What? Yes, this really happens, but Microsoft has kindly arranged it so that these three directives were not carried out on public API calls. For those of you who are struggling to get away from this (and have tried 'extern "C"' to no avail), use a .def file with an EXPORTS section. For those of you who are struggling with "The function was not loadable" errors in VEE, prepend an underscore to the beginning of the function name, and append an at sign and the number of bytes to the function name and the function will be loadable.

Getting back to the list, the callee cleans the stack. Parameter values have been pushed on the stack, and they need to be gotten rid of. Somebody has to do it, so stdcall says that it's the called function that is responsible for getting rid of them. It's an interesting detail, but of no consequence to us. Next however we have the function's return value is located in the EAX register upon return. Now, this is very important. The type of the function return value is shown at the left of the function declaration (a declaration is also what VEE calls a definition) and winds up stored in the variable on the left side of the equal sign when a function is used in an expression such as "bSuccess = WriteFile(...)". If the return value is not used it is simply overwritten. But wait, isn't something fishy here? Ponder this - it's important. What about functions such as:

```
DWORD GetCurrentDirectory(DWORD dwBufSize, LPTSTR
lpBuffer);
```

The SDK says that if the function succeeds, the return value is the number of characters copied to the buffer. What about lpBuffer? How does it get back to VEE? This is why you've been pondering. Besides the function return value, **the only way to return information from a function back to the caller is to pass one of the parameters by reference (address, pointer) and stuff the information into the memory pointed to by the pointer that has been passed.** This is so incredibly important it's worth saying a different way. If you want to return anything back to the caller besides a single number, then one of the parameters has to be passed by reference and that reference must point to a memory block big enough to store what you want to return.

Now I hear all you C people bitchin' and moaning that that's not true. The return value can itself be a pointer to whatever you want. Yes, that's so. And sometimes that's a keen way to do things except for one little detail: that pointer will have to have been allocated by something, and that something can only be the called function except in one instance. If that return value is not picked up by the caller and that memory is not later deallocated you have a memory leak and that's not cool.

That one instance is when the caller allocates memory and passes the pointer as a parameter. Well if you're going to do that then why bother returning the pointer as the return value when it can be put to far better use by returning a pass / fail value. Oh, the pointer may be reallocated you say? To make room for a bigger block if necessary? Well if you're going to do that then pass the blessed pointer itself by reference and return the new value in the same bloody parameter!

Ok, ok. There is one catch when doing this with VEE. If you're going to return values through parameters that have been passed by reference, then VEE dictates that you must retrieve the returned value via the parameter's output pin on the right side of the call object. Now, this is the reason I use Wrapper Functions. In the above GetCurrentDirectory example, the wrapper allocates a text variable large enough to contain any return text and calls the actual function with the buffer's size and a pointer to the buffer (remember, the function is defined with parameters (long, char*) so all I do is hook up the outputs of a formula that allocates & outputs the buffer and then outputs it's size to the input of the call object). The Return Value pin of the call object is sent to a global var where I can examine it later if necessary and the buffer output pin of the call object is sent to the output of the wrapper function. So outside the wrapper, to the rest of the program, I have effectively redefined:

```
DWORD GetCurrentDirectory(DWORD dwBufSize, LPTSTR
lpBuffer);
```
to
```
Text GetCurrentDirectory(void);
```

The reason I do all this is because then I can write strCurDir = GetCurrentDirectory(); instead of having several small objects plus one big honker scattered all over the place. When you start dealing with Windows programming, you're going to call lots of functions with lots of inputs and sometimes lots of outputs and do lots of things with lots of large data structures. When you do things with objects of this size the simple flowchart paradigm of VEE breaks down horribly and you wind up with a total mess.

The more compact it is the much, much easier to read it is. You can do traditional VEE if you want, but trust me: one little line of text is a whole lot better than using a 4"x6" area of precious screen real estate to simply accomplish one puny function call. I use wrapper functions and Massive Formula Objects (MoFos) because when I do so I can see one complete functional thought at once in one nice, neat little box instead of having that thought broken up over several different screens with all kinds of wires running all over the place. If your scroll bars are active then there's probably too much in one place, and you're going to have to troubleshoot and maintain all those wires running helter skelter all over the place too. A large VEE User Function or User Object can be a nightmare to maintain. Keep 'em small and keep 'em easy to figure out. The problem with that is you can't do a whole lot with Windows programming by keepin' 'em that way.

A&W Functions

There's one teeny weeny little thing to consider when doing what is essentially raw programming with the Windows API and that's ANSI and WideChar functions.

Originally (way back in the misty dawn of C programming), the American National Standards Institute defined the standard C library and several common practices to follow when programming in C. Though ANSI has long been replaced by the CCITT (which purportedly stands for "Can't Conceive Intelligent Thoughts Today) we still speak of standard C programming as "ANSI C". The ANSI defined a C string as an array of char terminated by a zero byte. That's all fine and good - if you want to speak only one language. Specifically English (because it also defined ASCII as the official C character set). A better way to represent strings had to be found, and UNICODE was the answer. A UNICODE character is a character with two bytes instead of one and is often called a "wide character". Ok, ok! I know it's not technically true that a UNICODE char is two bytes but for the purposes of the current discussion just pretend that it is.

Windows had already been under development for some time using ANSI only strings, but obviously it had to be able to accommodate wide characters too. The only way to provide wide character capability while not breaking software that uses one byte characters was to create two different versions of every function that deals with strings. This was eventually done and the result was that for every function that deals with strings there's one version appended with an "A" and one appended with a "W". The A functions are for one byte character string software and the W functions are for two byte character string software.

Whenever you're importing a function from almost any Windows API, if it deals with strings then you can bet that there's one A function and one W function. VEE is one byte character string software (we say it is not UNICODE aware), and it will always use A strings. So even though the SDK doesn't tell you about this, if there's a string parameter or return value then you'll have to append an "A" to the function name to have VEE find the function. There *are* exceptions to this rule, but when we find them I'll say something about it. In general, COM functions and Network functions do not have "A" versions. This is because COM was designed from the start to use wide characters and networks had been using wide characters long before Windows even knew networks existed.

As VEE moves into the future, I don't know what the status of it's UNICODE awareness will be. The upcoming release might mark the beginning of it's UNICODE aware self. As development is now being carried on in China, my guess would be that getting VEE into being multi-lingual would be an important priority. Like I said though, that's just a guess.

Where The Hell Are We?

To sum up, lets go over the most important points: Data has three attributes that must be considered when using the CFI. They are size, type and value. While data size must match on either side of the CFI, type need not and sometimes it actually either can't match or it's just easier to use a different type on one side or the other. Pointers point to data. The value of a pointer is the address of some data. A pointer can also point to a pointer. In this case, the value of the first pointer is the address of the value of the second pointer which then points to some data. An array is a pointer that points to a list of data entities, all of the same type. An array is passed over the CFI by declaring it as a pointer to the target type. Multidimensional arrays are handled exactly as if they had only one dimension.

A record is a group of discrete data handled as a unit. This is usually for convenience in handling data. Remember: an application lives *and dies* by it's data structures. Though a record can't be passed over the CFI, it *can* be passed by reference over the CFI. Well learn more about that in Type Spoofin' For Dummies, but it's best to take it a little at a time. The Front Page timer tells me that this page takes over 20 seconds to load and that means it's time to quit.

We saw several fascinating facts about calling convention, but in general we'll never have to deal with the ramifications of these except that one single value is returned from all compiled functions and if we want to return any more than that from the function then we have to pass one or more parameters by reference. For most function in the Windows API that deal with strings, there are actually two versions. One has an "A" appended and the other has a "W" appended. When calling Windows functions from VEE, we'll almost always be calling the "A" functions.