Some parts of an interface can change but others can't

devblogs.microsoft.com/oldnewthing/20141010-00

October 10, 2014

Raymond Chen

When I wrote about asking the compiler to answer calling convention questions, some people were concerned about whether this was a reliable mechanism or whether it was relying on something that can change in the future.

This is a special case of the question, "What parts of an interface can change, and what can't?" And it all boils down to compile-time versus run-time.

Assuming you are interested in binary compatibility (as opposed to merely source compatibility), then a decision made at compile-time can never be changed because the decision is already hard-coded into the application. For example, if you have a function that takes a parameter that is an enumeration, say,

```
enum FOO_OPTIONS
{
    FOO_HOP = 0,
    FOO_SKIP = 1,
    FOO_JUMP = 2,
};
```

then the values of `FOO_HOP`, `FOO_SKIP`, and `FOO_JUMP` are hard-coded into any program that uses them. The compiler will generate code like this:

```
; foo(FOO_JUMP);
    push 2
    call foo
```

Suppose you later change the header file to

```
enum FOO_OPTIONS
{
    FOO_HOP = 2,
    FOO_SKIP = 3,
    FOO_JUMP = 4,
};
```

Making a change in the new version of a header file has no effect on any existing programs which were compiled with the old version of the header file. There is no way for the `foo` function to tell whether the `2` it received as a parameter is a `FOO_JUMP` from the old header file or a `FOO_HOP` from the new one.

Therefore, you cannot reuse values in any existing enumerations or `#define`'s because the values are already compiled into existing programs. If you had given the value `2` different meanings in different versions of the header file, you would have in principle no way of knowing which header file the caller used. Of course, you can invent external cues to let you figure it out; for example, there may be a separate `set_foo_version` function that callers use in order to specify whether they are using the old or new header file. Of course, that also means that if there are multiple components that disagree on what version of `foo` they want, you have another problem.

Note that this is not the same as saying that the value of a symbol cannot change. We've seen this happen in the past with the `PSH_WIZARD97` flag, but these sorts of redirections are rare in practice.

Another thing that is hard-coded into an application is the calling convention. Once code is generated by the compiler to call a function, that's that. You can't change the calling convention without breaking existing code. That's why you can ask the compiler, "How would you call this function?" and trust the answer: If the compiler generates code to call the function using technique X (register set-up, what gets pushed on the stack first, *etc.*), then the function had better accept technique X in perpetuity. Of course, you need to be sure that what you observe is in fact all there is. There may be parts of the calling convention that are not obvious to you, such as the presence of a red zone or maintaining a particular stack alignment. Or it could be that the function is called only from within the module, and the compiler's whole-program optimization decided to use a custom nonstandard calling convention.

On the other hand, things determined at run-time can be changed, provided they are changed in a manner consistent with their original documentation. For example, the message numbers returned by `RegisterWindowMessage` can change because the documentation specifically requires you to call `RegisterWindowMessage` to obtain the message number for a particular named message.

If you want to know how to call a function, it's perfectly valid to ask the compiler, because at the end of the day, that's how the function gets called. It's all machine code. Whether that machine code was generated by a compiler or by an assembler is irrelevant.

Caveat: I'm ignoring whole-program optimization and link-time code generation, which allow the toolchain to rewrite calling conventions if all callers can be identified. We'll see more about this in a future article. The technique described in this article works best with

exported/imported functions, because it is not possible to identify all callers of such functions, and the compiler is forced to use the official calling convention. (You can also use it when inspecting .COD files for functions defined in a separate translation unit, for the same reason. That's the technique I used in the linked article.)

Raymond Chen

Follow