If you try to declare a variadic function with an incompatible calling convention, the compiler secretly converts it to cdecl

devblogs.microsoft.com/oldnewthing/20131128-00

November 28, 2013

Raymond Chen

Consider the following function on an x86 system:

```
void __stdcall something(char *, ...);
```

The function declares itself as `__stdcall`, which is a callee-clean convention. But a variadic function cannot be callee-clean since the callee does not know how many parameters were passed, so it doesn't know how many it should clean.

The Microsoft Visual Studio C/C++ compiler resolves this conflict by silently converting the calling convention to `__cdecl`, which is the only supported variadic calling convention for functions that do not take a hidden `this` parameter.

Why does this conversion take place silently rather than generating a warning or error?

My guess is that it's to make the compiler options `/Gr` (set default calling convention to `__fastcall`) and `/Gz` (set default calling convention to `__stdcall`) less annoying.

Automatic conversion of variadic functions to `__cdecl` means that you can just add the `/Gr` or `/Gz` command line switch to your compiler options, and everything will still compile and run (just with the new calling convention).

Another way of looking at this is not by thinking of the compiler as converting variadic `__stdcall` to `__cdecl` but rather by simply saying "for variadic functions, `__stdcall` is caller-clean."

Exercise: How can you determine which interpretation is what the compiler actually does? In other words, is it the case that the compiler converts `__stdcall` to `__cdecl` for variadic functions, or is it the case that the calling convention for variadic `__stdcall` functions is caller-clean?

Raymond Chen

Follow