What's the difference between int and INT, long and LONG, etc?

devblogs.microsoft.com/oldnewthing/20080325-00

March 25, 2008

Raymond Chen

When you go through Windows header files, you'll see types with names `INT`, `LONG`, `CHAR`, and so on. What's the difference between these types and the uncapitalized ones? Well, there isn't one any more. What follows is an educated guess as to the story behind these types. The application binary interface for an operating system needs to be unambiguous. Everybody has to agree on how parameters are passed, which registers are preserved, that sort of thing. A compiler need only enforce the calling convention rules at the boundary between the application and the operating system. When a program calls another function provided by that same program, it can use whatever calling convention it likes. (Not a true statement but the details aren't important here.) Therefore, a calling convention attribute on the declarations of each operating system function is sufficient to get everybody to agree on the interface. However, another thing that everybody needs to agree on is the sizes of the types being passed to those functions or used in structures that cross the application/operating system boundary. The C language makes only very loose guarantees as to the sizes of each of the types, so language types like `int` and `long` would be ambiguous. One compiler might decide that a `long` is a 32-bit integer, and another might decide that it's a 64-bit integer. To make sure that everybody was on the same page, the Windows header files defined "platform types" like `INT` and `LONG` with prescribed semantics that everybody could agree on. Each compiler vendor could tweak the Windows header file to ensure that the type definition for these platform types resulted in the value that Windows expected. One compiler might use `typedef long LONG` another might use `typedef __int32 LONG`. Okay, but this doesn't explain `VOID`. Maybe `VOID` was added for the benefit of compilers which didn't yet support the then-new ANSI C standard type `void`? Those older compilers could `typedef int VOID;` and functions that were declared as "returning `VOID`" would be treated as if they returned an integer that was always ignored. Or maybe it was just added to complete the set, who knows.

In the intervening years, most if not all compilers which target Windows have aligned their native types with Windows' platform types. An `int` is always a 32-bit signed integer, as is a `long`. As a result, the distinction between language types and platform types is now pretty much academic, and the two can be used interchangeably. New Windows functions tend to be introduced with language types, leaving platform types behind only for compatibility.

Raymond Chen

Follow