A __FILE__ Macro Which Does Not Contain the Whole Source File Path

20 Feb 2016

The `__FILE__` macro expands to the current source file name at compile time. It is not really useful if the source file paths which the build system uses, are very long, as this would bloat log output with long path names. It would be useful to have a shortened version, which only contains the file name without the whole path. This article describes how to implement a `__SHORT_FILE__` macro, that does not add any run time overhead.

Making Strings shorter at Compile Time

It is of course easy, to find the last slash in a string like `/home/user/src/project/src/file.cpp`, and return a pointer to the token `file.cpp`. This could be done at run time, and the overhead would most probably be neglibible in most thinkable situations, but it is no hassle to do it at compile time with C++11 using a `constexpr` function:

```
using cstr = const char * const;

static constexpr cstr past_last_slash(cstr str, cstr last_slash)
{
    return
        *str == '\0' ? last_slash :
        *str == '/'  ? past_last_slash(str + 1, str + 1) :
                       past_last_slash(str + 1, last_slash);
}

static constexpr cstr past_last_slash(cstr str)
{
    return past_last_slash(str, str);
}
```

This is certainly not the most elegant way to express a substring search. A nicer way would be to search for the last slash in a loop, or simply use library functions. In this context, however, it would not be possible to execute such code at compile time. This function in the presented form can be executed by a C++11 compiler at compile time, and then it is possible to directly embed the returned substring in the binary. (With C++14, it is possible to express this the easier to read loop way)

One could now write `printf` or `std::cout` (or whatever) to print just the file name via `past_last_slash(__FILE__)`, which is nice, but has two flaws:

1. It is still not as comfortable as a macro would be, i.e. `__SHORT_FILE__`
2. There is no guarantee, that the compiler would not call this function at runtime!

#1 can be fixed just by wrapping the function call into a `#define`, but that doesn't fix #2.

I wrote a small program which just does a `puts(__SHORT_FILE__)` using this *bad* macro, and it produced the following assembly output:

```
callq   __ZL15past_last_slashPKc ## past_last_slash(char const*)
movq    %rax, %rdi
callq   0x100000f58             ## symbol stub for: _puts
```

In order to obtain the short version of `__FILE__`, a function is called. This function is also called without parameters, which means that the compiler generated a function which is hard coded to this specific string. When compiling with `-O2`, this code disappeared, but there is no guarantee for that.

Important: *The return values of functions marked `constexpr` are only guaranteed to be calculated at compile time, if they are put into variables which are also marked `constexpr`.*

The following macro, which looks strange, fixes both:

```
#define __SHORT_FILE__ ({constexpr cstr sf__ {past_last_slash(__FILE__)}; sf_
_;})
```

This macro uses a `{}` scope, to instantiate a new helper variable on the fly, in order to force the return value of the helper function into a `constexpr` variable. At this point it is guaranteed, that the compiler will embed the return value into the binary, without generating a run time function call. The parentheses around that allow for transforming this scope block into an expression. `({int x = f(a, b); ...; x})` will just return the value of x, which was determined inbetween.

It is now possible to put this macro into any expression which would also accept `__FILE__` for logging, or printing. The assembly of the example program also looks better:

```
leaq    0x45(%rip), %rdi        ## literal pool for: "main.cpp"
callq   0x100000f54             ## symbol stub for: _puts
```

I found this macro pretty cool, as it reduces binary- and code size. And it even does this if optimization flags are disabled, in order to have a close look at the assembly while debugging.

Related Posts

Type List Compile Time Performance 25 Jun 2016

Generating Integer Sequences at Compile Time 24 Jun 2016

Executing Brainfuck at Compile Time with C++ Templates 16 Jun 2016