wnaabi Documentation

Release 0.1

Benjamin Kietzman

August 30, 2016

Contents

CHAPTER 1

__PRETTY_FUNCTION__

__PRETTY_FUNCTION__ (and its MSVC equivalent __FUNCSIG__) is a macro which expands to a string literal containing a nicely formatted representation of the function in which it was expanded. The interesting thing about __PRETTY_FUNCTION__ is its inclusion of template arguments.

Pace Boost.TypeIndex [1]: the original usage of this trick, so far as I am aware.

1.1 Examples:

MSVC outputs `const char *__cdecl prty<struct foo::bar>()`

```
template <typename T>
char const *prty() { return __FUNCSIG__; }

namespace foo { struct bar; }
int main() { std::cout << prty<foo::bar>() << std::endl; }
```

Clang outputs `const char *prty() [T = foo::bar]`

```
template <typename T>
char const *prty() { return __PRETTY_FUNCTION__; }

namespace foo { struct bar; }
int main() { std::cout << prty<foo::bar>() << std::endl; }
```

(the actual function used for typename retrieval is named `wnaabi::pretty_function::c_str`)

Using __PRETTY_FUNCTION__ this way enables access to the names of user defined types, complete with namespaces. This method can be used to retrieve strings for any type, but wnaabi uses other tricks to handle non-class types since the output of __PRETTY_FUNCTION__ is highly compiler dependent. Since the macro is expanded at compile time, `constexpr` functions can be used to trim the clutter.

`wnaabi::get_typename` returns the name of a user defined type without decoration by copying only the substring which contains the typename. Since the original string literal resulting from macro expansion is unreferenced, optimizing compilers can drop it from the resulting binary. The resulting buffer for `foo::bar` is exactly 8 `char`- not even the terminating null.

The string used to indicate residence in an anonymous namespace is compiler dependent (MSVC: `foo::'anonymous-namespace'::baz`, Clang: `foo::(anonymous namespace)::baz`, gcc: `foo::(anonymous)::baz`). These are filtered out by `wnaabi::get_typename`, yielding `foo::baz`. **NB:** wnaabi will probably never introduce a normalized anonymous scope string- `foo::__anonymous::baz`

[1] http://apolukhin.github.io/type_index/index.html

precludes the existence (in one translation unit, anyway) of `foo::baz` and `foo::__anonymous1::baz`. Therefore (IMHO), an anonymous scope string seems like pure bloat. For boffins who know when this would actually be handy/necessary, see this discussion of type-sorted polymorphic sets.

`__PRETTY_FUNCTION__` can be used to retrieve the name of an instantiation of a template class, but the arrangement of arguments following the template name is also compiler dependent. wnaabi uses `__PRETTY_FUNCTION__` only to retrieve the template name and defers formatting of template arguments to other functions. Although this allows wnaabi to guarantee the formatting of template types, it makes general support for non-type template parameters impossible.

Compiler	Macro documentation
MSVC	https://msdn.microsoft.com/en-us/library/b0084kay.aspx
Clang	not found - https://github.com/llvm-mirror/clang/blob/master/test/CodeGenCXX/predefined-expr.cpp
gcc	https://gcc.gnu.org/onlinedocs/gcc/Function-Names.html

1.1.1 References

1.1.2 Indices and tables

- genindex
- modindex
- search

CHAPTER 2

wnaabi

2.1 Who Needs an ABI?

This project presents a nifty implementation of `type_index` with some features you will love:

- Canonical typename format consistent across all compilers
- Access to typename **tokens** so you can hate on my canonical format however you want.
- Efficient creation, hashing, and typename retrieval- most of the work is done at compile time and this project doesn't even use the keyword `virtual`.

The premise of this project is that the C++ type system, generalized `constexpr` functions, and the rock-solid and minimalist C ABI can bootstrap a pretty awesome subset of the things a C++ ABI would be used for.

```
#include <cassert>
#include <wnaabi/type_info.hpp>

using namespace wnaabi;
namespace bar { struct baz; }

int main()
{
  assert(type_info<bar::baz>::name_tokens(runtime_visitors::stringify_t{}).str == "bar::baz");
}
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search