ztd.vargs

Release 0.0.0

ThePhD & Shepherd's Oasis, LLC

Dec 12, 2022

CONTENTS:

The premiere library for handling text in different encoding forms for C software.

CHAPTER

ONE

WHO IS THIS LIBRARY FOR?

Ideally?

Nobody.

This is mostly a proof of concept to make a proposal go through ISO/IEC JTC1 SC22 WG14 - Programming Language, C (AKA, C Standards Committee) easier and faster.

But if you're still curious, well, have a look! It works mostly like `va_start`, `va_arg`, and `va_end`, but so far iterates over all arguments (regardless of whether they were passed statically or not).

Listing 1: How snazzy and cool, 3 arguments iterated over and used!

```
#include <ztd/vargs.hpp>
#include <ztd/idk/assert.hpp>

int imul3(...) {
        ztdc_va_list vl;
        ztdc_va_start(vl);
        int num0 = ztdc_va_arg(vl, int);
        int num1 = ztdc_va_arg(vl, int);
        int num2 = ztdc_va_arg(vl, int);
        ztdc_va_end(vl);
        return num0 * num1 * num2;
}

int main() {
        int result0 = imul3(3, 4, 5);
        ZTD_ASSERT(result0 == 60);
        return result0;
}
```

1.1 API Reference

Each given function aids in iterating over a `ztdc_va_list`. Currently, the lists are specified to iterate over **all** arguments where possible, unless compiler optimization or similar interferes with the documented ABI and API constraints of a particular compiler / platform combination. This means that for the following:

Listing 2: An example piece of code with 2 named arguments that go unused.

```
#include <ztd/vargs.hpp>
#include <ztd/idk/assert.hpp>

double dmul3([[maybe_unused]] int a,
             [[maybe_unused]] double b,
             ...) {
        ztdc_va_list vl;
        ztdc_va_start(vl);
        int num0    = ztdc_va_arg(vl, int);
        double num1 = ztdc_va_arg(vl, double);
        int num2    = ztdc_va_arg(vl, int);
        ztdc_va_end(vl);
        return num0 * num1 * num2;
}

int main() {
        double result = dmul3(3, 4.5, 5);
        ZTD_ASSERT(result == 67.5);
        return 0;
}
```

Even the named argument `a` and `b` will be iterated over. (A future revision of this library may correct for this, but it is how it is for now.) This means that you should avoid using this when there are arguments present in the list, and therefore wish to execute ``

See the *architecture list* for supported architectures.

1.1.1 Functions

typedef struct *ztdc_va_list* **ztdc_va_list**
: The `va_list` type. Can be used in any scenario where the argument list is empty.

Remark

Currently, only C++ supports such a declaration: Standard C compilers will break on it.

---

ztdc_va_start(_VL)
: Initializes and starts up the iteration of a `...` argument list!

Remark

This currently will iterate over arguments that are already presents in the non-variable arguments part of the call, so factor this in appropriately if used with a mix of statically-known and variable arguments!

Mandates

- The _VL parameter must have been previously initialized by a call to ztdc_va_start.

Cursed? This call may not work well for everything, since occasionally critical information is missing from just the raw function call. Prefer ztdc_va_arg_in, which takes both a ztdc_va_list and the name of the function it is within.

Parameters

- **_VL** – **[in]** A ztdc_va_list (not a pointer to one!).

`ztdc_va_start_in`(_VL, ...)

Initializes and starts up the iteration of a `...` argument list!

Remark

This version uses specialist information from the function prototype to properly adjust the internal implementation. This is important for functions which return large structs that are placed in special positions thanks to Return Value Optimization (RVO), Indirect Struct Return Optimizatino (ISRO), and other behaviors specific to a given platform/ABI/compiler architecture.

Mandates

- The _VL parameter must have been previously initialized by a call to ztdc_va_start.
- The `...` token parameters must form a complete, non-overloaded function name (qualified or unqualified) which can have its type (`decltype(__VA_ARGS__)`) taken. The behavior is undefined if this is not the function that is actually calling this.

Parameters

- **_VL** – **[in]** A ztdc_va_list (not a pointer to one!).
- **...** – **[in]** The function name this is being called from.

`ztdc_va_arg`(_VL, _TYPE)

Initializes and starts up the iteration of a `...` argument list!

Remark

This currently will iterate over arguments that are already presents in the non-variable arguments part of the call, so factor this in appropriately if used with a mix of statically-known and variable arguments!

Mandates

- _TYPE shall not be a reference type (pointer types are fine).
- The _VL parameter must have been previously initialized by a call to ztdc_va_start.

Cursed?

Parameters

- **_VL** – **[in]** A ztdc_va_list (not a pointer to one!).

- **_TYPE** – **[in]** The type to pass in. Must not be a reference type.

ztdc_va_end(_VL)
Ends the iteration of a ztdc_va_list.

Mandates

- The _VL parameter must have been previously initialized by a call to ztdc_va_start.

Cursed?

Parameters

- **_VL** – **[in]** A ztdc_va_list (not a pointer to one!).

1.2 Supported Architectures

Each architecture needs to be supported explicitly, with occasional builtins or other things aiding in iteration and work. If you'd like to contribute an implementation, please make a patch to the repository!

Furthermore, it is imperative to note the various circumstances this can appear under in C++ includes member functions. Unfortunately, we have not yet developed a way of knowing this information, and member functions do change the way the compiler interacts with the ABI and where it places arguments.

Table 1: Architecture List

Compiler	Architecture	Supported? "Notes / Documentation"	
Microsoft Visual C++	x86_64 (AMD64)		*Notes*
	x86 (i686)		
	ARM		
	ARM64		
Clang	x86_64 (AMD64)		
	x86 (i686)		
	ARM		
	ARM64		
GCC	x86_64 (AMD64)		
	x86 (i686)		
	ARM		
	ARM64		

1.2.1 Microsoft Visual C++ - x64

The VC++ x64_86 implementation relies on the System V ABI that Microsoft uses for it's calling convention, particularl for it's Variable Argument functions. It lays out much of it's details in two documents:

- x64 Calling Conventions
- x64 Stack Usage

As the storage in registers in fickle, we explicitly rely on 2 things to keep work going:

1. an intrinsic from <intrin.h> called `_AddressofReturnAddress()`; and,
2. the fact that we can use assembler to access registers that get spilled to the stack.

The second is not necessarily guaranteed: highly-optimized Variable Argument function calls do **not** spill values to the stack. But in general, registers documented such as `rcx` and `rdx` are frequently used when any amount of work is done inside of these functions, and that causes the stack to immediately "re-home" the values in those registers to (8-byte aligned) places just above the address of the return address (e.g., what we will use as our stack pointer to walk the stack for arguments).

Note: *Observed (But Not Documented)*

In testing, any usage of `ztdc_va_list` and `ztdc_va_start` within a Variable Argument function triggered the register rehoming. This allowed us to get at the arguments that were previously in registers that were both in practice and in documentation too widely reused, too volatile, and too hot to reliably extract.

Note that simply walking the stack is not 100% effective, even with stack re-homing: floating point arguments are not typically re-homed in the Microsoft System V ABI, and therefore must be accessed directly in their registers `xmm0` through `xmm3` for the corresponding to the first 4 arguments.

- **For the first 4 arguments:**
  - Non-floating point types including integers, aggregates (`std::is_aggregate_v` and all C types) with `sizeof(Type) <= 8`; `rcx`, `rdx`, `r8`, and `r9`. Re-homed to locations `rsp + 8`, `rsp + 16`, `rsp + 24`, `rsp + 32`, `rsp` representing the address from `_AddressofReturnAddress()`.
  - Floating point types, `float` and `half`, and all `__mNN` types up to `__m64`: `xmm0`, `xmm1`, `xmm2`, `xmm3`. Not re-homed to anywhere on the stack reliably.
  - All other values are turned into pointers, and those pointer values are stored in the `rcx`, `rdx`, `r8`, and `r9` (and re-homed).
- **For each argument after:**
  - Stored on the stack from `rsp + 40` onwards, regardless of whether or not any registers are re-homed. The way they are stored follows the above: direct values for all types that are `sizeof(T) <= 8`, and pointers to said values for anything else.

Warning: There seem to be alignment issues on Windows that are not clearly explained in the documentation. `rsp` and the "rehomed" space may not be aligned properly, despite the documentation stating that non-leaf (framed) functions must be aligned properly. It is hard to get it to keep the data in the right place and occasionally seems to produce data pointers in the rehomed and and other stack pointer places that are not where they are expected to be.

A cheap check for knowing if you have walked off the edge of the stack is testing if the pointer value for any stack values is greater than the address of the `ztdc_va_list` list. This can be done as an assert (which can be turned off in Release builds by-default).

`float` types are automatically promoted to `double`, and so if a person requests `float` it must be converted to `double` first and then explicitly downcast within the platform's implementation of `ztdc_va_next`.

1.3 Licenses, Thanks and Attribution

`ztd.vargs` is dual-licensed under either the Apache 2 License, or a corporate license if you bought it with special support. See the LICENSE file or your copy of the corporate license agreement for more details!

1.3.1 Thank You

A special Thanks to Alex Gilding for writing N2584, which spurred the research and discovery into this subject!

CHAPTER

TWO

INDICES & SEARCH

2.1 Index

INDEX

Z