P1631R0: Object detachment and attachment

Document #:	P1631R0
Date:	2019-06-16
Project:	WG21 Programming Language C++
	WG21 SG1 Concurrency study group
	WG21 SG12 Undefined behaviour study group
	WG21 SG14 Low latency study group
Reply-to:	Niall Douglas
	<s_sourceforge@nedprod.com>

[*Note:* This edition of P1631R0 is a quickly written temporary stand-in paper substituting for the proper paper cowritten by Bob Steagall and Niall Douglas, which was delayed due to an automobile incident. The proper paper will probably replace this paper within a few days of the mailing. – end note]

This paper outlines a proposal to add the following operations to the C++ abstract machine, which provide an additional way of beginning and ending object lifetime, in addition to those already described in [basic.life]:

1. Object *detachment*, which is the one-way, in-place, cast of a live object into an array of bytes representing its detached object representation, ending the lifetime of the input object, and beginning the lifetime of the returned array of bytes.

2. Object *attachment*, which is the one-way, in-place, cast of a byte array representing a previously detached live object, into an instance of the original live object. The lifetime of the input byte array ends, and the lifetime of the reattached object begins.

Additionally, an additional utility function is proposed which prevents dead store elimination for a range of bytes. This can be inefficiently implemented in today's compilers using compiler-specific assumptions, however a compiler built-in intrinsic would be more desirable.

It is believed that these proposed changes are sufficient to implement the transmission of object represenations of a subset of C++ types, without undefined behaviour, through memory shared between concurrent processes, memory mapped in from another device by DMA, process bootstrap from a database of shared binary Modules, and the elemental operations for implementing zero-copy serialisation and deserialisation. One also gains object relocation in memory, as byte arrays are trivially copyable, so one can change the memory location of an object by detaching it, bit copying it, and reattaching it, for objects of the subset of C++ types which are detachable and attachable.

These changes ought to also finally enable operating system kernels, and other fixed latency code, to no longer disable strict aliasing optimisation, if they refactor their current reinterpret cast based code to use detach and attach casts instead.

An early draft of this proposal was presented to the May 2019 WG14 C programming language committee meeting in London. It was checked against the current C2x working draft's memory and object model, and was found to be compatible with no changes required to the existing wording (though most present C compilers implement these operations very inefficiently). It was discovered during discussion that the potential for dead store elimination would need to be handled in order to ensure that detaching objects in shared memory would work correctly, and thus an additional dead store elimination prevention function was added since the WG14 N2367 paper draft.

A partial reference implementation of `detach_cast()` and `attach_cast<T>()` based on bit casting can be found at https://github.com/ned14/quickcpplib/blob/master/include/detach_cast.hpp.

A reference implementation of the proposed standard library support can be found at https://github.com/ned14/quickcpplib/blob/master/include/in_place_detach_attach.hpp.

Both of these reference implementations have been deployed into the reference library implementation for [P1031] *Low level file i/o*, where the proposed API design has been found to work well.

Changes since D0 draft 1/WG14 N2367 (published to WG14):

- Purged the pointer provenance stuff.
- Purged the enhanced memory and object model stuff.
- Reworked the definition of proposed detach and attach cast to meet WG14 feedback.
- Added ensure stores, as it was pointed out by WG14 that it is unavoidable in order to correctly implement shared memory.

1 Acknowledgements

My thanks to the WG14 C programming language committee for reviewing during their London meeting what was essentially a C++ proposal paper, and giving such useful feedback which resulted in a rearchitecture of this proposal.

Thanks to Jens Gustedt and Martin Uecker for their feedback on an early edition of D0, and further feedback at the WG14 meeting.

2 References

[N4034] Pablo Halpern,
Destructive Move
https://wg21.link/N4034

[P0023] Denis Bider,
Relocator: Efficiently moving objects
https://wg21.link/P0023

[P1031] Douglas, Niall
Low level file i/o library
https://wg21.link/P1031

[P1144] Arthur O'Dwyer, Mingxin Wang
Object relocation in terms of move plus destroy
https://wg21.link/P1144