D1631R0 draft 2/Dxxxx: Enhanced C/C++ memory and object model

Document #:	WG21 D1631R0 draft 2/WG14 N2367
Date:	2019-03-25
Project:	WG14 Programming Language C++ WG21 Programming Language C++ WG21 SG12 Undefined behaviour study group

Reply-to:	Niall Douglas <s_sourceforge@nedprod.com>

An enhanced memory and object model for C++ and C based around implementing a subset of [P1434] *Discussing pointer provenance* to make much more rigorous the modification of memory, and adding two new core operations to objects:

1. *Detachment*, the reinterpretation of a live object into an array of bytes representing that object.
2. *Attachment*, the reinterpretation of a previously detached object representation into a live object.

It is believed that these changes are sufficient to implement 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, and substantially enhanced default move implementations which can use CPU registers for object transport.

It should be emphasised that care has been taken to ensure that none of the proposed changes prevent the implementation of standard C++ on even very small embedded devices, or on heavily concurrent architectures such as GPUs.

Changes since draft 1:

- Simplified the introduction.
- Standardesed attachment and detachment.
- Rewrote, removed or expanded upon contentious stuff which early reviewers objected to.

Contents

1 Introduction

Hardware memory management units, page-faulted virtual memory, shared memory, memory mapped files and process concurrency have been ubiquitous on the major platforms, and on many of the embedded ones, for over two decades, yet despite that C++ implementations rely heavily on these features, the C++ standard has no knowledge nor support for them. This renders much contemporary C and C++ into undefined behaviour territory, which in turn substantially limits the extent to which C++ implementations can perform stronger optimising transformations to code. This proposal would remedy that situation, by making input/output – i.e. a subset of deserialisation/serialisation – well defined.

As messing significantly with the standard memory and object model ought not to be done more frequently than every twenty years, I have taken the opportunity to strengthen the reasoning that C++ implementations can make about memory and objects, sufficiently that an implementation could *formally prove* the correctness of how a C++ program treats memory. The proposal would

be that ambiguous and correctness-unproveable code would no longer compile by default in new C++ standard enabled compilers, but an opt-in 'escape hatch' would enable non-conforming code to compile under the old rules, albeit with those sections of code marked as unproveable. I have relied heavily on the work of Sewell et al on developing a formally provable C memory model, please see [P1434] *Discussing pointer provenance* for more detail.

All this would make defined behaviour some common things done on today's computing systems which are currently undefined or implementation defined behaviour:

- Memory mapped files.
- Shared memory.
- Direct Memory Access (DMA).
- More than one C/C++ program running at a time.
- A C/C++ program being executable a second time after its first time, being able to change its behaviour based on accessing state stored during the first time.
- Page fault allocated memory.
- Copy-on-write memory pages.

That in turn makes possible standardising the following proposed facilities in [P1031] *Low level file i/o* which are currently not possible in the current C/C++ memory and object model.

- `section_handle`

  Wraps private, anonymous (disposed on last handle close) or named shared memory (i.e. a file on the filing system).

- `map_handle`

  Wraps the mapping of a portion of a `section_handle` into the memory of the local process, implementing the synchronous read/write API of `io_handle` as `memcpy()`.

- `mapped_file_handle`

  Combines a `section_handle` and a `map_handle` into a 'fire and forget' `file_handle` implementation, which transparently uses memory maps under the hood to implement a `file_handle` implementation complete with barriers, byte range locking, synchronisation etc.

- `map_view<T>`

  A *non-owning* view of a memory map, refines `span<T>`.

- `mapped<T>`

  Combines a `section_handle` and a `map_handle` into an *owning* view of a memory map, refines `map_view<T>`.

2 Motivation and Scope

2.1 Genericise static data initialisation

The major C++ implementations already make use of page-fault copy-on-write driven object initialisation in the form of the static initialisation of global objects with constexpr constructors. During compilation, the compiler will compute what the byte representation would be for such objects – which is possible thanks to the constexpr constructor – and stores that byte representation in the final executable binary. During static data initialisation, those objects can simply be blessed into life, with no runtime code execution necessary. The first write into such objects causes a page fault, and the kernel will make a private copy of that page, replacing the shared read-only page mapped in from the program executable.

This proposal makes generic that process, such that objects can be *attached* and *detached* from the C++ program at any time. The program, under this proposal, can perform attachment at any time during its execution.

This makes it possible for future binary Module objects to be dynamically attached into a C++ program, and detached if that Module implements detachment for itself. This would enable the future implementation of shared library support into C++.

2.2 One C++ program only

I have deliberately restricted the scope of considering multiple C++ programs to no more than multiple instances of the current C++ program, whether executed more than once over time, or more than once concurrently, possibly over a network. What this proposal does not cover:

- The same source code compiled into more than one C++ program.
- The same source code compiled for more than one architecture, or settings, or configuration.
- Non-identical C++ programs.

I appreciate that some will be disappointed by this. All I can say is that this proposal is already huge. Baby steps!

2.3 No inter-process communication

Early reviewers of this paper have remarked on the lack of proposed mechanism for the multiple C++ programs to communicate with one another. Thus, this proposal specifies that it is well defined for C++ programs to exchange detached storage instances with one another, but says nothing about how those C++ programs would indicate to one another when one program has detached an instance, and another program is able to attach that same instance.

Early reviewers find this situation to be problematic, chiefly because one cannot state a complete, cradle-to-grave, order of sequencing between object detachment in one C++ program through to

object attachment in a different C++ program. Some find this to be a showstopper, however there are good reasons to just not go there yet.

Firstly, [P0668] *Revising the C++ memory model* hasn't shaken fully out yet. As that paper points out, the current C++ memory model is not efficiently implementable on Power and nVidia architectures, so the C++ 20 standard model has been made more complex in order to better match hardware realities with the memory model (e.g. simply happens before/strongly happens before). The current memory model does not require hardware assisted implementations of preventing read and write reordering, however code written to use this model would perform extremely poorly without a hardware assist. And it must be borne in mind that process-based, rather than thread-based, concurrency often has no hardware assist, thus leading to coarse rather than fine synchronisation granularity for process-based over thread-based concurrency. As an example, if a CPU needs to synchronise a detached storage instance with a program running on the other side of a network connection, it would need to transmit that storage instance to the other side before it can indicate to the other process that it can proceed. This makes synchronisation exceptionally costly relative to processing, thus coarse synchronisation granularity is the only reasonable.

Secondly, there is a lot of movement currently occurring in the hardware space in this area, which makes standardising an interprocess communication mechanism unwise in my opinion at the current time. In my opinion, it would be better to let operating system kernels evolve how best they will support synchronising new hardware such as persistent RAM first, see how that falls out, and then seek to standardise whatever emerges.

This incomplete proposal will be unsatisfying to all. However, operating system kernels provide a wide suite of interprocess communication mechanisms upon which the proposed enhancements can be coordinated. That situation is little different to today's situation, so little is lost. In my opinion best to bite off what we can chew.

3 Design decisions, guidelines and rationale

3.1 C compatibility

I think it important that these very fundamental changes ought to be 100% compatible with C such that implementations can retain as similar as possible a memory and object model for both C and C++.

3.2 Differences from [P1434] *Discussing pointer provenance*

That paper proposes that every storage instance has an identifier unique throughout the program execution, and that object lifetime is the same as storage instance lifetime such that that the identifier will mutate every time an object begins life, even if at the same location in memory. This permits run-time enforcement of pointer provenance, and thus prevents a wide class of memory-related bugs.

This paper divorces the lifetime of storage instances from the objects they contain, and thus the unique identifier of the storage instance is no longer that of the specific incarnation of the objects within it. The unique identifier of the storage instance is furthermore unique across all possible executing C++ program instances, such that one C++ program may transmit a storage instance identifier to another, and the other C++ program will be able to attach the same storage instance.

Thus this paper's provenance checking model is rather weaker than P1434's, because it is per-storage-instance, not per-object-incarnation. Some may feel this is a fatal flaw. However, equally, runtime checking of every object incarnation would make such checked implementations infeasible to deploy in production, whereas it may be the case that runtime checking of storage instances, if combined with a compile-time static analysed checker of object provenance, might produce production deployable executable binaries.

4 Proposed enhanced memory and object model

4.1 Memory model (6.6.1)

1 The fundamental storage unit in the memory model is the *byte*. A byte is at least large enough to contain any member of the basic execution set and the eight-bit code units of the Unicode UTF-8 encoding form, and is composed of a contiguous sequence of bits, the number of which is implementation-defined[1]. The least significant bit is called the *low-order bit*; the most significant bit is called the *high-order bit*. The ~~memory~~ storage available to a C++ program consists of one or more sequences of contiguous bytes (arrays). Every byte ~~has a unique address~~ can be uniquely identified across all *reachable C++ programs* using the identifier of its *storage instance* and its offset into that instance.

2 [*Note:* The representation of types is described in X.X. – end note]

3 A 'reachable C++ program' can be defined by the implementation as one of the following options:

1. The currently running C++ program only. In this definition, all modifications to storage instances are lost when the C++ program's execution ends, which would suit embedded devices without persistent storage.
2. Sequential executions of the unmodified current C++ program over time, where at least one modification to storage instances by one execution is made available to subsequent executions of the same C++ program, so long as each execution forms a total sequential ordering. This definition would suit embedded devices with a single CPU and some persistent storage.
3. Concurrent executions of many instances of the current C++ program, where modified storage instances can be passed between those concurrently executing instances, including across heterogeneous compute[2].

[1] There is a strong argument that future C++ ought to standardise the eight bit byte, leaving non eight bit byte architectures on older C++ standards. This argument is much weaker for C, but still worth considering.

[2] **Be very clear** that non-identical C++ programs are not supported in this proposal (though it is left open that implementations may offer extended guarantees).

4 A *storage instance* is a contiguous array of bytes in which an object or array of objects starts and ends its lifetime. If reachable by other C++ programs[3], every storage instance has a unique identifier across all those reachable C++ programs, made up of the identifiers of its *memory pages*[4], its offset into the first of those memory pages, and the type of the object(s) stored in that storage instance[5]. Previously *detached* objects within reachable storage instances may be *reattached*, and possibly *re-detached*[6] once again, from the C++ program.

5 A *memory page* is an architecture-determined grouping of bytes. There can be one or more sizes of memory page on an architecture, with restrictions on alignment or granularity. Some architectures fix the *memory location* of each memory page, others permit a memory page to have one or more memory locations. Memory pages come in the following kinds:

1. Private, anonymous pages visible only to the current C++ program. The contents of these are always discarded when the current C++ program ends execution. Storage instances kept in this kind of memory page are not required to have a unique identifier.
2. Copy-on-write pages which always have the same initial content upon first object attachment, but in which modifications are local to the current C++ program only, and those modifications are always discarded when the current C++ program ends execution. Storage instances kept in this kind of memory page are not required to have a unique identifier.
3. Pages modifications of which are potentially visible to other reachable C++ programs. Storage instances kept in this kind of memory page **are** required to have a unique identifier.

6 Memory pages can be concurrently accessible to more than one reachable C++ program at a time, however accessing a memory page outside an attached storage instance is not defined. It is not defined what occurs if a storage instance is attached to more than one reachable C++ program at a time.

7 A *memory location* is either an object of scalar type, or a maximal sequence of adjacent bit-fields all having nonzero width , referring to an offset within a memory page. [*Note:* Various features of the language, such references and virtual functions, might involve additional memory locations that are not accessible to programs but are managed by the implementation. – end note] Two or more threads of execution can access separate memory locations without interfering with each other. It is therefore not defined what occurs if more than one memory location refers to the same offset within the same memory page[7].

8 [*Note:* Thus a bit-field and an adjacent non-bit-field are in separate memory locations, and therefore can be concurrently updated by two threads of execution without interference. The same applies to two bit-fields, if one is declared inside a nested struct declaration and the other is not, or if the two are separated by a zero-length bit-field declaration, or if they are separated by a non-bit-field

[3] Private, anonymous memory is by definition unreachable by other programs, however implementations may give such storage instances a unique identifier anyway, in order to implement a run-time validator of memory usage correctness.

[4] For example, the device, inode and offset into the file backing the storage of the storage instance.

[5] A high-quality hash of the type-id of the type is suggested.

[6] There are objects whose detached representation can only ever be attached, and cannot be detached. There are also objects which can never be attached nor detached.

[7] This allows the compiler to assume that the same file will never be mapped into more than one location in the same process, otherwise all memory would have to be assumed could alias.

declaration. It is not safe to concurrently update two bit-fields in the same struct if all fields between them are also bit-fields of nonzero width. – end note]

9 [*Example:* A class declared as

```
struct {
  char a;
  int b:5,
  c:11,
  :0,
  d:8;
  struct {int ee:8;} e;
}
```

contains four separate memory locations: The member `a` and bit-fields `d` and `e.ee` are each separate memory locations, and can be modified concurrently without interfering with each other. The bit-fields `b` and `c` together constitute the fourth memory location. The bit-fields `b` and `c` cannot be concurrently modified, but `b` and `a`, for example, can be. – end example]

4.2 Object model (6.6.2)

10 The constructs in a C++ program create, destroy, attach, detach, refer to, access and manipulate objects. An *object* is created by a definition (X.X), by a *new-expression* (X.X), by attachment, when implicitly changing the active member of a union (X.X), or when a temporary object is created (X.X). An object occupies a ~~region of storage~~ storage instance in its period of construction (X.X), throughout its lifetime (X.X), during detachment (X.X), and in its period of destruction (X.X). [*Note:* A function is not an object, regardless of whether or not it occupies storage in the way that objects do. – end note] The properties of an object are determined when the object is created. An object can have a name (Clause X). An object has a storage duration (X.X) which influences its lifetime (X.X). An object has a type (X.X). Some objects are polymorphic (X.X); the implementation generates information associated with each such object that makes it possible to determine that object's type during program execution. For other objects, the interpretation of the ~~values~~ array of bytes found therein is determined by the type of the *expressions* (X.X) used to access them.

(Trivially obvious changes next until ...)

11 An object of trivially copyable, standard-layout , trivially attachable, or trivially detachable type (X.X) shall occupy contiguous bytes of storage.

(Only trivially obvious changes remain in this section)

4.3 Object detachment and attachment (new subsection)

12 The operation of *detachment* shall be the rendering of a live object into its *detached object representation* which shall be an array of `byte` equal in number to the `sizeof` the live object. Upon

successful detachment, the lifetime of the object or objects within the portion of the storage instance detached shall end.

13 The operation of *attachment* shall be the rendering of a previously detached object representation into a live object. Upon successful attachment, the lifetime of the object or objects within the portion of the storage instance attached shall begin. It shall be implementation defined what occurs if a non-reachable C++ program attaches a previously detached object representation.

14 An object of type `T` is said to have *non-vacuous detachment* if there exists a free function `span<byte> in_place_detach(span<T>) throws`[8] for objects of that type, or one of its subobjects. This user-defined function renders an array of live objects into their detached object representations. The cast operator `span<byte> detach_cast(span<T>)` shall be available within the implementation body of `in_place_detach()` functions, this reinterprets the array of `T` into an array of `byte`; unlike `reinterpret_cast`, it shall be an error to access the input array of `T` after the cast. Implementations shall carry a dependency from the input array of `T` to the array of `byte`[9].

15 An object of type `T` is said to have *non-vacuous attachment* if there exists a free function `span<T> in_place_attach(span<byte>) throws` for objects of that type, or one of its subobjects. This user-defined function renders an array of bytes representing array of detached object representations into an array of live `T` objects. The cast operator `span<T> attach_cast(byte<T>)` shall be available within the implementation body of `in_place_attach()` functions, this reinterprets the array of `byte` into an array of `T`; unlike `reinterpret_cast`, it shall be an error to access the input array of `byte` after the cast. Implementations shall carry a dependency from the input array of `byte` to the array of `T`.

16 An object of type `T` shall have *trivial* attachment or detachment if it is neither a pointer nor reference type, and does not contain subobjects which are of pointer or reference type, and does not have non-vacuous attachment or detachment. Types with trivial attachment or detachment, and without a non-vacuous attachment or detachment implementation, shall have a *default* compiler-generated `in_place_attach(span<byte>)` or `in_place_detach(span<T>)` implementation which shall be defined as the calling of `detach_cast(span<T>)` for detachment, and `attach_cast(byte<T>)` for attachment.

4.4 Object and reference lifetime (6.6.3)

17 The *lifetime* of an object or reference is a runtime property of the object or reference. An object is said to have *non-vacuous initialization* if it is of a class or array type and it or one of its subobjects is initialized by a constructor other than a trivial default constructor. [*Note:* Initialization by a trivial copy/move constructor is non-vacuous initialization. – end note] The lifetime of an object o of type `T` begins when:

18 - ~~storage~~ a storage instance with the proper alignment and size for type `T` is obtained, and one of

19 - if the object has non-vacuous initialization, its initialization is complete, or

[8] We assume the presence of [P0709] *Zero overhead deterministic exceptions* throughout this proposal.

[9] i.e. aliasing between the two regions cannot occur, and memory modifications cannot be reordered in a visible way across the cast.

20 • if the object has non-trivial attachment, its attachment routine is complete,

except that if the object is a union member or subobject thereof, its lifetime only begins if that union member is the initialized member in the union (X.X), or as described in X.X. The lifetime of an object *o* of type `T` ends when:

21 • if `T` is a class type with a non-trivial destructor (X.X), the destructor call starts, or

22 • if `T` is a class type with a non-trivial detachment (X.X), the detachment call starts, or

23 • the ~~storage~~ storage instance which the object occupies is released, or is reused by an object that is not nested within `o` (X.X).

24 The lifetime of a reference begins when its initialization is complete. The lifetime of a reference ends as if it were a scalar object requiring storage.

25 [*Note:* X.X describes the lifetime of base and member subobjects. – end note]

(Only trivially obvious changes remain in this section)

4.5 Indeterminate values (6.6.4)

(Only trivially obvious changes are in this section)

4.6 Storage duration (6.6.5)

26 The *storage duration* is the property of an object that defines the minimum potential lifetime of the storage instance containing the object. The storage duration is determined by the construct used to create the object and is one of the following:

27 • static storage duration

28 • thread storage duration

29 • automatic storage duration

30 • dynamic storage duration

31 Static, thread and automatic storage durations are associated with objects introduced by declarations (X.X) and implicitly created by the implementation (X.X.X). The dynamic storage duration is associated with objects created by a *new-expression* (X.X.X.X).

32 The storage duration categories apply to references as well.

33 When the duration of a storage instance begins, pointers representing the address of any part of that storage instance may be created. These pointers gain a *live provenance* associated with that particular storage instance.

34 [*Note:* A pointer to the byte after the end of a storage instance gains a *dead provenance* associated with that particular storage instance. It **may** *also* have a live provenance with a separate storage instance. A pointer cannot have more than one live provenance at a time, but it may have a live and one or more dead provenances simultaneously. – end note]

35 When the end of duration of a ~~region of storage~~ storage instance is reached, ~~the values of~~ all pointers representing the address of any part of that ~~region of storage~~ storage instance ~~become invalid pointer values~~ gain dead provenance (X.X.X). ~~Indirection through an invalid pointer value and passing an invalid pointer value to a deallocation function have undefined behavior. Any other use of an invalid pointer value has implementation-defined behavior.~~

36 Indirection through pointers has an associated contractual precondition that provenance is live (X.X). Therefore indirection through pointers with dead provenance may cause a contract violation for violation of precondition if the implementation has been configured to do so (X.X). Any other use of an invalid pointer value has implementation-defined behavior.

(Trivially obvious changes follow until 6.6.5.4.3 Safely-derived pointers, the whole of which is to be replaced with:)

4.6.1 Defined pointer usage

37 A non-void, non-null, non-function pointer object will have an associated *provenance*:

38 - It may have zero or one *live* provenance to a byte somewhere within the valid byte array which makes up its associated storage instance.

39 - It may have zero or many *dead* provenances to associated storage instances, which can be identified by their identifier unique across all reachable C++ programs.

40 A non-null function pointer object will have a provenance unique to the function it points to.

41 If the provenance of one or more externally supplied non-void, non-null, non-function pointer object cannot be determined, they all shall be assumed to have the provenance of a single storage instance of an array of the type of the pointer[10].

42 The following operations involving pointers are defined:

43 - Comparison of two null pointers, which shall be considered equal.

44 - Comparison of one null pointer to a nonnull pointer, which shall be considered unequal.

45 - Ordered comparisons (<, <=, >, >=), where the provenance is common.

46 - Pointer arithmetic, so long as the resulting pointer addresses a byte within its provenanced storage instance (whereupon it shall be live if its storage instance is live), or the single byte immediately after (whereupon it shall be dead). The new pointer retains its source's provenance.

47 - Indirection, if provenance is live.

48 - Comparisons, including equality and inequality, of two void pointers.

[10] This prevents the compiler having to maintain runtime provenance tracking information for every pointer passed to a function, or used from global state. It also means that all pointers to the same type passed to a function, including globals used, are assumed could alias one another, as is the case in the current standard.

[49] The following operations involving two non-void, non-null pointers shall fail to compile with a diagnostic:

[50] - Comparisons, including equality and inequality, of two pointers without common provenance (user should convert to `void*` beforehand).

5 Non-standardese proposed changes

We shall break away from Standardese now, as these are more fluid due to implementation questions.

5.1 Utility functions

5.1.1 `span<byte> memory_page_sizes(span<byte> tofill, span<T> storage_instance) noexcept`

This routine fills an array of bytes with the two-power shifts needed to construct the memory page size of each of the memory pages in which a storage instance is stored. If the input array is too small, an empty span is returned.

This call can be implemented easily on Microsoft Windows, Mac OS and BSD using existing APIs. I am unaware of a straightforward technique on Linux, though a trivially simple device driver, or new syscall, can implement it within a dozen lines of code using kernel APIs.

It should be noted that the bitscan operation necessary to calculate the bitshift of each page size is not fast on low end CPUs, though it needs to be done only once per memory page size. It should also be noted that we assume a two's power size, which as far as I am aware is true for any hardware on which modern C++ can be implemented. It finally should be noted that this function is unavoidably racy in the presence of multiple threads of execution, unless all threads are synchronised to the execution of this call, which is best done by the end user.

5.1.2 Polymorphic object detach and attach

This will be controversial, but I would propose the following attachment and detachment primitives to enable polymorphic objects to be detached and reattached.

- `void std::in_place_polymorphic_attach(T*) noexcept`[11]

  This would make valid into the current C++ program a `T` polymorphic object. It would be called by a `span<T> in_place_attach<T>(span<byte>) throws` implementation as the final operation to give life to a just-attached `T` object.

  For `vptr` based implementations, this would rewrite the vptr to point at the static metadata for that type of polymorphic object. I cannot think of a non-vptr implementation which could not be similarly made valid.

[11] Why a pointer not a reference? A reference ought to be to a live object, whereas pointers can refer to any memory location.

- `void std::in_place_polymorphic_detach(T*) noexcept`

  This would make invalid in the current C++ program a `T` polymorphic object. It would be called by a `span<byte> in_place_detach<T>(span<T>) throws` implementation as the first operation to remove life from an about-to-be-detached `T` object.

  For `vptr` based implementations, this would rewrite the vptr to be null, thus guaranteeing that any attempt to use the object polymorphically will fail.

I have left open the question as to whether polymorphic types, which otherwise meet the trivially attachable or detachable criteria, ought to be considered trivially attachable. My personal opinion is that this is tricky. We don't implement trivially copyable for polymorphic types, despite that on vptr implementations it would work just fine. On that basis, we should not make it automatic here either.

However, there is an argument that all the known C++ 20 implementations are vptr-based, as was the case for all known C++ 17 implementations. So one could argue that 'vptr has won', and to go ahead and make a new enhanced category of types called 'bitwise copyable' types, which is a superset of trivially copyable types. See [P1029] *SG14 [[move_relocates]]* for some ideas of what this enhanced category of bitwise-copyable types could make possible.

5.2 Static initialisation, attachment and detachment

Currently, non-local static initialisation occurs before `main()` begins by the execution of the constructors of all objects stored at global level. For objects with trivial or constexpr constructors, the compiler *may* precompute the live byte representation of those objects at compile or link time, thus avoiding the need to call the constructors for those objects when the program is executed.

It would be proposed that this process be retermed into attachment and detachment instead. Objects with trivial or constexpr construction placed into static storage initialised at process start would now be implemented as-if as follows:

```
byte temp[sizeof(T)];
new(temp) T(static init args ...);  // T constructor is constexpr or trivial
span<byte> p = in_place_detach(o);  // or detach_cast(o) if in_place_detach() not implemented
memcpy(executable_binary, p.data(), p.size());
```

This is a special case for backwards compatibility, as `T` may contain pointers or references and thus not be trivially attachable and detachable. Yet we would silently call `detach_cast()` if `in_place_detach()` is not available for this type. This is safe, as the linker fixes up any constexpr statically initialised pointers and references to be correct.

5.3 Escape hatching from defined pointer usage

The proposed well defined usage of pointers is sufficiently strict that almost no existing code using pointers would compile, and would have to be refactored. Consider the following:

```
A *a;  // non-null
B *b;  // non-null

// This compiles
B *x = (B *)((uintptr_t) a + offsetof(A, the_b));

// But this does not (x does not have the same provenance as b)
assert(b == x);

// Neither does this (x has no provenance)
*x;
```

Under the proposed text, integers cannot have provenance, and thus `x` has no provenance.

There are three ways out:

1. You must manually specify which provenance a newly created pointer must take during casting to pointers:

```
B *x = (B *[a])((uintptr_t) a + offsetof(A, the_b));
```

2. You permit compilers to track provenance during pointer => integer => pointer conversions, for which it is hard to conclusively write out a set of comprehensive rules.
3. You add a formal escape hatch like Rust's `unsafe` blocks in which the existing pointer usage rules apply. The entire block is marked as 'unproveable' by the compiler.

Even if you add significant complexity to compiler provenance tracking of temporaries such that single input provenance code like the above does compile, there still will be tens of millions of lines of code which will need adjusting to be less ambiguous.

I am unaware of any proposal before WG21 or WG14 which would break backwards source compatibility so profoundly. It has never been done before. Yet the gains are immense, and perhaps even crucial to the long term survival of the C ecosystem as the expectations of software reliability increase, yet the annual improvements to hardware capabilities decrease.

5.4 C language equivalents

It is proposed that C gains underscore-capital mirrors of the above functions, though perhaps in non-span form. For example, the C mirror of `span<T> in_place_attach<T>(span<byte>) throws` might be `T *_InPlaceAttach(char *) fails`[12].

6 Frequently asked questions

6.1 How is object relocation implemented using this proposal?

To relocate an object's memory location, one would do the following:

[12] This assumes the C equivalance of [P0709] along the lines of [P1095] *Zero overhead deterministic failure.*

1. Detach the object into its detached byte representation.
2. Cause the relocation of those bytes to a different address e.g. `memcpy()`, or page table modification e.g. `mmap()`.
3. Reattach the object from its relocated byte representation.

As trivially attachable and detachable types have default attachment and detachment implementations in this proposal, that would mean that all types without pointers and references within them can now be relocated. This substantially expands the number of types whose move constructor and move assignment could be defaulted, and whose representation can be transported in CPU registers instead of on the stack.

This would make object relocation proposals such as [P1144] *Object relocation in terms of move plus destroy* considerably easier to implement.

7 Acknowledgements

Thanks to Jens Gustedt and Martin Uecker for their feedback.

8 References

[N4800] *Working Draft, Standard for Programming Language C++*
https://wg21.link/N4800

[P0593] Richard Smith,
Implicit creation of objects for low-level object manipulation
https://wg21.link/P0593

[P0668] Hans-J. Boehm, Olivier Giroux, Viktor Vafeiades and others,
Revising the C++ memory model
https://wg21.link/P0668

[P0709] Herb Sutter,
Zero-overhead deterministic exceptions
https://wg21.link/P0709

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

[P1029] Douglas, Niall
SG14 [[move_relocates]]
https://wg21.link/P1029

[P1095] Douglas, Niall
Zero overhead deterministic failure – A unified mechanism for C and C++
https://wg21.link/P1095

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

[P1434] Hal Finkel, Jens Gustedt, Martin Uecker,
Discussing Pointer Provenance
https://wg21.link/P1434