P1631R1: Object detachment and attachment

Document #:	P1631R1
Date:	2019-07-11
Project:	WG21 C++ Programming Language
Audience:	SG1 Concurrency study group
	SG12 Undefined behaviour study group
	SG14 Low latency study group

Reply-to: Niall Douglas <s_sourceforge@nedprod.com>
Bob Steagall <bob.steagall.cpp@gmail.com>

[*Note:* This paper remains a work in progress. Much of it is unfinished as you will notice, but due to lack of available time, we were not able to do better before the Cologne meeting. We apologise, but enough detail is here that people will get the gist of the proposal, and be able to say whether to abandon this direction, or continue work in this direction. Note that this proposal does not supplant the need for most of what is proposed in [P0593] *Implicit creation of objects for low-level object manipulation*, as one still needs to bless and unbless the byte arrays the operators proposed here work with. – end note]

Abstract

This paper proposes adding two new operations to the C++ abstract machine, *object detachment* and *object attachment*, intended to provide a new tool for finer-grained control of object lifetime. Two new utility functions are also proposed to prevent dead store elimination for a user-specified range of bytes, and to require memory reloading for a user-specified range of bytes.

Changes since N2367 (published to WG14):

- Removed the discussion of pointer provenance.
- Removed the discussion of enhanced memory and object model.
- Revised the definition of proposed detach and attach cast to incorporate WG14 feedback.
- Added `ensure_stores()`, as it was determined by WG14 that such a facility is required to implement shared memory correctly.
- Added `ensure_loads()`, which Paul McKenney pointed out is also required to implement shared memory correctly.

Contents

1 Introduction **3**

2 Motivating problems **4**
2.1 Shared memory . . . 4
2.1.1 Linux io_uring . . . 4
2.1.2 Problem for the current C++ abstract machine . . . 5
2.2 Memory-mapped files . . . 8
2.2.1 Background and recent relevant evolution in the standard . . . 8
2.2.2 Object detachment . . . 9
2.2.3 Object attachment . . . 9
2.2.4 Customisation points . . . 11
2.3 Object transmission . . . 14
2.4 Object relocation . . . 15
2.5 Existing practice . . . 16

3 Proposed solution **16**
3.1 Design constraints and rationale . . . 16
3.2 Object detachment/attachment . . . 16
3.2.1 Detachable types . . . 16
3.2.2 Detach cast . . . 16
3.2.3 Attach cast . . . 16
3.2.4 Ensuring stores . . . 16
3.2.5 Ensuring loads . . . 16

4 Discussion **16**
4.1 Alternatives . . . 16

5 Proposed Wording **17**

6 Acknowledgments **17**

7 References **17**

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 many of the embedded ones, for well over two decades. And yet, despite the fact that virtually all C++ implementations rely heavily on these features, the C++ abstract machine has no awareness of them, nor operations to directly support them. This relegates much contemporary C++ into the realm of undefined behavior if it wishes to guarantee runtime efficiency, as standard C++ implementations cannot guarantee that the optimiser will always elide the memory copies otherwise required for defined behaviour. [P0593] *Implicit creation of objects for low-level object manipulation* would remove the need to memory copy implicit lifetime types mapped in from a file, and this proposal further eliminates the need to copy memory for all other kinds of C++ objects mapped in from files.

Our core proposal is to add two new operations to the C++ abstract machine:

1. *Object detachment*, which is the one-way, in-place cast of a live object into a corresponding array of bytes representing its detached state without invoking a destructor. This cast ends the lifetime of the input object, and begins the lifetime of the array of bytes.

2. *Object attachment*, which is the one-way, in-place cast of a byte array containing a previously detached object into a live instance of the original object's type. This cast ends the lifetime of the input byte array, and begins the lifetime of the reattached object without invoking a constructor.

An early draft of this proposal was presented to the May 2019 WG14 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 (although at present most C compilers implement these operations very inefficiently).

During those discussions it was discovered that reload optimisation and dead store elimination could silently cause unexpected errors when detaching objects stored in shared memory. To address this problem, two new utility functions are proposed to prevent reload optimisation and dead store elimination for a user-specified range of bytes. Such functionality can be implemented today using compiler-specific assumptions and facilities, albeit somewhat inefficiently. Built-in compiler intrinsics, with the performance gains they could bring, would be much more desirable.

We believe the changes proposed in this paper are sufficient to transmit the object representations of detachable types, without undefined behavior, through memory shared by concurrent processes, memory-mapped files, memory mapped from a device via DMA, network connections, and elemental operations for implementing zero-copy serialization and deserialization. One also gains relocation in memory for the subset of objects which are both detachable and attachable, as byte arrays are trivially copyable. One could, for example, change the memory location of a detachable object by detaching it, performing a bitwise copy to another byte array, and reattaching it using the copy of the array.

These proposed changes should enable operating system kernels, and other fixed latency code, to finally reap the benefits of globally enabling strict aliasing optimization, assuming they refactor their current `reinterpret_cast`-based code to use the proposed detach and attach casts instead.

2 Motivating problems

2.1 Shared memory

Reading from and writing to memory shared between processes, or even directly with the kernel, is becoming an increasingly first choice for 'bare metal' interprocess communication. Let us examine a read world use case of this, and why it is problematic for current C++.

2.1.1 Linux io_uring

Linux kernel v5.0 adds a new low latency i/o interface based on the direct reading and writing of memory shared between the process and the kernel, thus entirely avoiding the overhead of system calls for i/o. The core 'io_uring' structure is this:

```
struct io_uring_sqe {
  __u8 opcode;                 // The operation to perform
  __u8 flags;
  __u16 ioprio;
  __s32 fd;                    // The file descriptor
  __u64 off;                   // The offset into the file if relevant
  __u64 addr;                  // Address of scatter-gather list
  __u32 len;                   // Items in scatter-gather list
  union {
    __kernel_rwf_t rw_flags;
    __u32 fsync_flags;
    __u16 poll_events;
  };
  __u64 user_data;             // Passed through to completion notification
  union {
    __u16 buf_index;
    __u64 __pad2[3];
  };
};                             // Exactly 64 bytes in length, expected to be one per cache line
```

To schedule an i/o, one writes an instance of the above structure into a shared, unidirectional, ring buffer, and updates the ring buffer tail index to indicate that new i/o requests have been issued. Apart from the need to issue memory fences, this is a simple writing to a C structure:

```
1: sqe->opcode = IORING_OP_READV;
2: sqe->fd = fd;
3: sqe->off = 0;
4: sqe->addr = &iovec;
5: sqe->len = 1;
6: sqe->user_data = some_value;
   write_barrier();                  /* ensure previous writes are seen before tail write */
7: sqring->tail = sqring->tail + 1;
   write_barrier();                  /* ensure tail write is seen */
```

If configured to use polling rather than blocking, the above is literally all one has to do to schedule a read from or write to a device on Linux using the io_uring interface.

To find out when the i/o has completed, one reads from a second unidirectional ring buffer filled by the kernel and drained by the process which consists of this structure:

```
struct io_uring_cqe {
  __u64 user_data;             // Passed through from the submission
  __s32 res;                   // i/o result, positive if bytes transfered, negative if an error code
  __u32 flags;
};                             // Exactly 16 bytes in length
```

One reads this exactly the same way as the kernel reads from the submission ring buffer, by checking the tail index, and if new completions have arrived, to drain them:

```
1: unsigned head = cqring->head;
   read_barrier();
2: if (head != cqring->tail) {
3:   unsigned index = head & (cqring->mask);  // head modulus ring buffer entries
4:   struct io_uring_cqe *cqe = &cqring->cqes[index];
     /* process completed cqe here */
     ...
     /* we've now consumed this entry */
5:   head++;
   }
6: cqring->head = head;
   write_barrier();
```

As you can see, from Linux 5.0 onwards one can perform device i/o exclusively using simple copies of memory. All one needs to do is:

1. Issue read and write reordering barriers to both the compiler and CPU to ensure that reads and writes do not become reordered nor merged in a way which breaks the correct sequence for the shared memory inter-process communication channel.
2. Prevent the compiler from eliminating apparently-to-it dead stores. As [N4455] *No Sane Compiler Would Optimize Atomics* reports, atomics are subject to dead store elimination, so this requires the use of `volatile` in the current C++ standard.
3. Cause the compiler to reload object values at certain points, despite that from the compiler's perspective, those values could not possibly have changed since the last time that they were read. This also requires the use of `volatile` in the current C++ standard.

If you want lots more detail on how io_uring works, please see `http://kernel.dk/io_uring.pdf`.

2.1.2 Problem for the current C++ abstract machine

Up until now, in some ways the C++ abstract machine has 'gotten away' with not having to concern itself with objects in memory spontaneously changing their value i.e. programmers do not currently have to mark all data which goes to and from a syscall as `volatile`. It could do this because a system call to the kernel (e.g. `read()`), or at the very least a call to an extern function in the C library (e.g. `pthread_mutex_lock()`), would force the compiler to assume that values may have escaped to other threads of execution, and thus they must be written out before, and reloaded after, the syscall or extern function call.

Leaving aside the fact that this 'one size fits all' escape analysis induces poor codegen for i/o which increasingly is heading towards hundreds of nanoseconds in completion time (see later in this paper), the problem for polling-based io_uring code is that the compiler cannot know when it must write out or reload state (the read and write barriers affect scope of reordering only, not optimisations like dead store elimination). So in current C++, one has the choice of either marking pointers to io_uring structures as `volatile` (and being very careful to always access the structures through the volatile pointer), or marking all the members of the structure as `volatile`. This does not produce high quality codegen, because loads and stores cannot be combined, reordered, merged and elided. Moreover, the compiler cannot know that the `sqe->addr` refers to a scatter list which represents buffers written to by the kernel, so *all of those* must **also** be accessed using volatiles, which is a severe anti-optimisation, and may not be possible if the scatter list is processed by a third party library.

Using the `ensure_loads()` and `ensure_stores()` functions proposed by this paper, this is solved:

```
1: sqe->opcode = IORING_OP_READV;
2: sqe->fd = fd;
3: sqe->off = 0;
4: sqe->addr = &iovec;
5: sqe->len = X;
6: sqe->user_data = some_value;
   sqe++;

   ... write N more io_uring_sqe structures

   // Prevent dead store elimination to these bytes, AND perform a release barrier
   ensure_stores(sqe_first, (sqe_last - sqe_first) * sizeof(*sqe),

                 memory_flush_none,      // do not flush modified cache lines to main memory

                 memory_order_release);  // prevents reads and writes to this region preceding
                                         // this operation being reordered to after this operation

7: sqring->tail = sqring->tail + 1;      // Can't use atomics here, they are subject to dead
                                         // store elimination

   ensure_stores(&sqring->tail, sizeof(sqring->tail),
                 memory_flush_none,
                 memory_order_release);
```

Upon completion:

```
   // atomic load after store of same memory location could be elided, so
   // force the compiler to (this one time) load cqring->head fresh
   ensure_loads(&cqring->head, sizeof(cqring->head),

                 memory_flush_none,     // do not force reading of these bytes from main memory

                 memory_order_acquire); // prevents reads and writes to this region subsequent
                                        // to this operation being reordered to before this

   // Use atomic load to prevent reordering of subsequent operations before this
```

```
1: unsigned head = atomic_load(&cqring->head, memory_order_acquire);
   unsigned firstindex = UINT_MAX, lastindex = UINT_MAX;
2: if (head != cqring->tail) {
3:   firstindex = head & (cqring->mask);  // head modulus ring buffer entries
4:   for(auto h = head, index = firstindex;
         head != cqring->tail && index >= firstindex;
         index = ++head & (cqring->mask)) {
5:     lastindex = index;
     }
     // Ensure the loading of all ring buffer entries from firstindex to lastindex
     ensure_loads(&cqring->cqes[firstindex],
                  (lastindex - firstindex + 1) * sizeof(struct io_uring_cqe),
                  memory_flush_none,
                  memory_order_acquire);
6:   while(firstindex <= lastindex) {
7:     struct io_uring_cqe *cqe = &cqring->cqes[firstindex++];
       ... process cqe ...
8:     head++;
     }
   }
9: cqring->head = head;
   // Prevent dead store elimination to these bytes, AND perform a release barrier
   ensure_stores(&cqring->head, sizeof(cqring->head),
                 memory_flush_none,
                 memory_order_release);
```

The key difference from `volatile struct io_uring_sqe *` and `volatile struct io_uring_cqe *` – which would be the traditional way of implementing this in standard C++ – is that the compiler can batch, reorder, merge and elide the structure reads and writes in between the barriers, or use auto-vectorisation upon them, or any other substantial as-if optimising transformation. This would be unimportant for half a dozen structures, but becomes important for hundreds of structures, such as for high frequency socket i/o, or lots of small block i/o to a very high performance storage device e.g. non-volatile RAM.

Moreover, down the line, if we are to implement an optimal-efficiency replacement for `iostream` (see [P1026] *A call for a Data Persistence (`iostream` v2) study group*), it is currently believed that zero-copy serialisation and deserialisation of types will involve the compile-time generation of long sequences of small-block scatter-gather buffers for each serialisable type, with a buffer emitted for each valid part of a structure (i.e. padding bytes are explicitly left out). These would be optimisable by the compiler under current (embryonic) plans (i.e. can be merged, folded, reordered, elided), but it would be even better again if the compiler has the opportunity to use SIMD-based batch scatter-gather memory copying[1] i.e. that `volatile` is not getting in the way, and forcing individual and separate copies of every individual structure member.

[1]For example, ARM NEON lets you load every N-th 32-bit quanity, thus letting one de-interleave the actual data format.

2.2 Memory-mapped files

2.2.1 Background and recent relevant evolution in the standard

Memory mapped files are very widely used in contemporary C++ codebases, yet most uses of memory mapped files falls into undefined behaviour. This is because `mmap()` returns a pointer to memory which does not contain objects with lifetime in the C++ abstract machine, so any use of such storage without first constructing objects into it is undefined behaviour. As existing C++ code almost always reinterpret casts the pointer returned by `mmap()` into a C++ type, and then uses that object instance immediately without calling a constructor, that creates a problem for well defined use of memory mapped files in C++.

[P0593] *Implicit creation of objects for low-level object manipulation* proposes a new operation `std::bless()` which can create C++ objects from bytes of unknown provenance without the calling an object constructor for the type of object, for a subset of possible C++ types. The subset of C++ types which can be legally blessed into valid lifetime are called *implicit lifetime types*, which are those types satisfying[2]:

- `std::is_aggregate<T>`, OR
- `std::is_trivially_destructible<T>` AND one of:
  - `std::is_trivially_default_constructible<T>`
  - `std::is_trivially_copy_constructible<T>`
  - `std::is_trivially_move_constructible<T>`

`std::bless()` handles the making live of objects of implicit lifetime types mapped in from a file. P0593R4 does not propose a specific operation to handle the generation of that mapped file in the first place (i.e. to legally end the lifetime of object written into a mapped file), but by email Richard proposed this as suitable:

```
void unbless(void *p, size_t n) {
  new (p) char[n];
}
```

This relies on implicit lifetime types being required to have trivial destructors, so reusing its storage for new objects ends the lifetime of the objects previously therein. One could thus write objects to a memory mapped file, and unbless them before unmapping the file in order to ensure that the C++ abstract machine does not think that objects with defined values still exist in the unmapped memory.

The proposed legal ability to use memory mapped files is welcome, however the limitations upon the the kinds of object which you can use are not. This paper proposes additional lifetime operations to bring in a wider subset of C++ types for legal usage within memory mapped files: `attach_cast<T>()` and `detach_cast()`.

[2]Note that these requirements are weaker than `TriviallyCopyable`.

2.2.2 Object detachment

This paper proposes a new operation for the abstract machine, one which immediately ends the lifetime of the input object, but not through calling its destructor. Instead, the lifetime of a new array of byte exactly the same size as the input object is begun, and it is required for the implementation to ensure that the *detached byte representation* returned is sufficient for later reattachment of an identical object value by the current C++ program. Detach cast has the following signature:

```
//! \brief A reference to a byte array sized the same as ‘T‘
template <class T> using byte_array_reference = byte (&)[sizeof(T)];

//! \brief A const reference to a byte array sized the same as ‘const T‘
template <class T> using const_byte_array_reference = const byte (&)[sizeof(T)];

/*! \brief Detaches a live object into its detached byte representation, ending the
lifetime of the input object, and beginning the lifetime of an array of byte sized
exactly the size of the input object at the same memory location, which is returned.
All references to the input object become INVALID. Any use of the input object after
detachment has occurred is illegal!
*/
template<class T>
[[nodiscard]] constexpr inline byte_array_reference<T> detach_cast(T &) noexcept;

template<class T>
[[nodiscard]] constexpr inline const_byte_array_reference<T> detach_cast(const T &) noexcept;
```

It is required that the memory location of the returned byte array equals the memory location of the input object. It is required that the size of the returned byte array is exactly that of the input object.

If the input type is polymorphic, it is required that the dynamic type of the object is rendered useless in the current program. For a vptr-based implementation, overwriting the vptr with a null pointer would suffice.

2.2.3 Object attachment

This operation performs the reverse of object detachment. It takes in a previously detached object representation *which was detached by the current C++ program*, ends the lifetime of that input byte array, and begins the lifetime of a new object. Attach cast has the following signature:

```
/*! \brief Reattaches a previously detached object, beginning the lifetime of the
output object, and ending the lifetime of the input array of byte.
All references to the input byte array become INVALID. Any use of the input array
after attachment has occurred is illegal!
*/
template<class T>
[[nodiscard]] constexpr inline T &attach_cast(byte_array_reference<T> &) noexcept;

template<class T>
[[nodiscard]] constexpr inline const T &attach_cast(const_byte_array_reference<T> &) noexcept;

/*! \brief Convenience overload for attach_cast which accepts a ‘void *‘ instead
```

```
of a byte array reference. Helps end users avoid having to reinterpret cast.
*/
template<class T>
[[nodiscard]] constexpr inline T *attach_cast(void *) noexcept;

template<class T>
[[nodiscard]] constexpr inline const T *attach_cast(const void *) noexcept;
```

It is required that the memory location of the returned object equals the memory location of the input byte array. If the type being reattached is polymorphic, during attach casting it is required that the dynamic type is reset to the appropriate metadata for the type being attached. For a vptr-based implementation, setting the vptr to the value for which an object of this type would have would suffice.

It is undefined behaviour if:

1. If the current C++ program did not detach the object now being reattached.
2. The type of attach cast is not **identical** to the **originally constructed type** of the object which was previously detached.

Using these new operations, one can now use objects of any C++ type within memory mapped files with defined behaviour:

```
void *mem = mmap(fd);
// mem currently points to storage without objects with lifetime

// Construct a std::string object into the storage pointed to by mem
string *a = new(mem) string("hello!");

// Detach the std::string object at a into its detached byte array
// representation using P1631 proposed detach_cast(), ending the
// lifetime of std::string, and beginning the lifetime of a byte
// array exactly overlaying the storage of the input std::string
// with the same byte values as the representation of the input
// std::string
byte_array_reference<string> &b = detach_cast(*a);

// It is now an abstract machine violation to dereference a, which
// as a pointer to a dead object, now has an indeterminate value

// Ensure dead store elimination to the detached byte array does
// not occur using P1631 proposed ensure_stores()
ensure_stores(&b, sizeof(b));

// Tell the abstract machine that the lifetime of the byte array at
// b has ended, and any further reads of that byte array is a violation.
// If we did not have the ensure_stores() immediately preceding this,
// this would invoke dead store elimination going right back up to
// the placement new operation.
unbless(&b, sizeof(b));

// Unmap mem, which removes the storage of the byte array entirely
// Any accesses to this region will result in SIGSEGV, and thanks
// to unbless(), a static analyser can
// issue an error at compile time about any subsequent reads of mem
// without needing to understand anything about munmap()
munmap(mem)

...

// Map fd back in again, likely to a different memory location
void *mem = mmap(fd);

// Reattach previously detached std::string, beginning the lifetime
// of a new std::string object from the input byte array
string *s = attach_cast<string>(mem);

// Use the string, it is a perfectly normal std::string
cout << *s << std::endl;

// Destroy the string object, releasing any dynamically allocated
// storage
s->~string();
```

2.2.4 Customisation points

An immediate observation is that types cannot customise detachment and attachment with type-specific handling. This paper proposes two customisation points which allow developers to specify custom handling, and through convention, any code which needs to perform object detachment and attachment would use these ADL discovered free functions instead of the direct cast operations:

```
  /*! \brief An ADL customisation point for the in-place detachment of an array of
  live ‘T‘ objects into an array of bytes representing their detached object representations.
  If the input is const T, the output is const bytes.
  */
  template<class T>
  constexpr inline auto in_place_detach(span<T> objects) noexcept
  {
    using byte_type = typename std::conditional<std::is_const<T>::value, const byte, byte>::type;
    span<byte_type> ret;
    if(objects.empty())
    {
      ret = span<byte_type>((byte_type *) objects.data(), (byte_type *) objects.data());
      return ret;
    }
    for(size_t n = 0; n < objects.size(); n++)
    {
      auto &d = detach_cast(objects[n]);
      if(0 == n)
      {
        ret = {d, sizeof(d)};
      }
      else
      {
        ret = {ret.data(), ret.size() + sizeof(d)};
      }
    }
  return ret;
}
//! Convenience overload taking a single value
template<class T>
constexpr inline span<byte_type> in_place_detach(T &object) noexcept;

/*! \brief An ADL customisation point for the in-place attachment of previously detached
object representations, back into an array of live ‘T‘ objects.
If the input is const byte, the output is const T.
*/
template<class T>
constexpr inline span<T> in_place_attach(span<byte> bytearray) noexcept
{
  span<detail::byte_array_wrapper<T>> input = {(detail::byte_array_wrapper<T> *) bytearray.data(),
      bytearray.size() / sizeof(T)};
  span<T> ret;
  if(bytearray.empty())
  {
    ret = span<T>((T *) bytearray.data(), (T *) bytearray.data());
    return ret;
  }
  for(size_t n = 0; n < input.size(); n++)
  {
    T &a = attach_cast<T>(input[n].value);
    if(0 == n)
    {
      ret = {&a, 1};
    }
    else
    {
      ret = {ret.data(), ret.size() + 1};
    }
  }
  return ret;
}
//! Convenience overload taking a single value
template<class T>
constexpr inline T &in_place_attach(span<byte> bytearray) noexcept;
//! Convenience overload taking a void *
template<class T>
constexpr inline T &in_place_attach(void *mem) noexcept;

/*! \brief An RAII refinement of ‘span<T>‘ for automatically calling ‘in_place_attach()‘
and ‘in_place_detach()‘ on an input array of ‘T‘. Move-only, detaches only on final
object destruction.
*/
template <class T> class attached : protected span<T>
{
  using _base = span<T>;

public:
  //! The index type
  using index_type = typename _base::index_type;
  //! The element type
  using element_type = typename _base::element_type;
  //! The value type
```

```cpp
  using value_type = typename _base::value_type;
  //! The reference type
  using reference = typename _base::reference;
  //! The pointer type
  using pointer = typename _base::pointer;
  //! The const reference type
  using const_reference = typename _base::const_reference;
  //! The const pointer type
  using const_pointer = typename _base::const_pointer;
  //! The iterator type
  using iterator = typename _base::iterator;
  //! The const iterator type
  using const_iterator = typename _base::const_iterator;
  //! The reverse iterator type
  using reverse_iterator = typename _base::reverse_iterator;
  //! The const reverse iterator type
  using const_reverse_iterator = typename _base::const_reverse_iterator;
  //! The difference type
  using difference_type = typename _base::difference_type;

public:
  //! Default constructor
  constexpr attached() {}  // NOLINT

  attached(const attached &) = delete;
  //! Move constructs the instance, leaving the source empty.
  constexpr attached(attached &&o) noexcept : _base(std::move(o)) { static_cast<_base &>(o) = {
      nullptr, 0}; }
  attached &operator=(const attached &) = delete;
  constexpr attached &operator=(attached &&o) noexcept
  {
    this->~attached();
    new(this) attached(std::move(o));
    return *this;
  }

  //! Detaches the array of `T`, if not empty
  ~attached()
  {
    if(!this->empty())
    {
      in_place_detach(as_span());
    }
  }

  //! Implicitly construct from anything for which `in_place_attach<T>()` is valid.
  template<class U>
  requires { in_place_attach<T>(std::declval<U>())) }
  constexpr attached(U &&v)
      : _base(in_place_attach<T>(static_cast<U &&>(v)))
  {
  }
  //! Explicitly construct from a span of already attached objects
  explicit constexpr attached(adopt_t /*unused*/, _base v)
      : _base(v)
  {
  }

  //! Returns the attached region as a plain span of ‘T‘
  constexpr _base as_span() const noexcept { return *this; }

  using _base::empty;
  using _base::first;
  using _base::last;
  using _base::size;
  using _base::size_bytes;
  using _base::ssize;
  using _base::subspan;
  using _base::operator[];
  using _base::operator();
  using _base::at;
  using _base::begin;
  using _base::cbegin;
  using _base::cend;
  using _base::crbegin;
  using _base::crend;
  using _base::data;
  using _base::end;
  using _base::rbegin;
  using _base::rend;
  using _base::swap;
};
```

Some may find it surprising that there are array-based customisation points. These turn out to be very useful in practice, and the `attached<T>` RAII manager, which is most frequently used when working with memory mapped files, falls out naturally from the array-based design.

2.3 Object transmission

Combining the facilities above, one can now transmit many kinds of C++ objects through shared memory without introducing undefined behaviour:

```cpp
struct AbstractType
{
  virtual void print() = 0;
};
struct ImplType : AbstractType
{
  int value;
  virtual void print() override { std::cout << value << std::cout; }
};
...

void send(shared_memory<byte> mem, AbstractType value)
{
  // Grab the dynamic type of the input value
  const char *typeidstring = typeid(value).name();

  // Detach the value into its detached byte representation
  span<byte> bytespan = in_place_detach({&value, 1});

  // Copy the span of bytes into the shared memory
  mem.push_back(typeidstring, bytespan);

  // Signal the other side that new objects have landed
  mem.notify();
}

std::shared_ptr<AbstractType> receive(shared_memory<byte> mem)
{
  // Retrieve the next detached object in the shared memory
  auto [typeidstring, bytespan] = mem.pop_front();

  if(0 == strcmp(typeidstring, typeid(ImplType).name()))
  {
    span<ImplType> values = in_place_attach<ImplType>(bytespan);
    auto ret = std::make_shared<ImplType>(std::move(values[0]));
    // Attached type has non-trivial destructor, must call to avoid UB
    values[0].~ImplType();
    return ret;
  }
  else if ...
}
```

Under this proposal, only the current C++ program can transmit and receive objects via shared memory with defined behaviour. This would seem to be so limited as to be useless, however given the almost limitless variety of ways of configuring a C++ program to be memory-layout incompatible with another (e.g. macros, structure packing, compiler options), even for identical source code, it is infeasible to define behaviour for anything more than the current C++ program.

(It *may* be possible in the future to widen 'the current C++ program' definition to include multiple, concurrent, instances of the same C++ program binary. This is a big task, requiring its own proposal paper)

2.4 Object relocation

Something interesting in the Object transmission example is that one is also relocating an object value in memory i.e. changing its memory location. Indeed, object value relocation is as simple as:

```
template<class T>
constexpr inline T &relocate_object(void *dest, T &obj) noexcept
{
  span<byte> src = in_place_detach(obj);
  memmove(dest, src.data(), sizeof(T));
  return in_place_attach<T>(dest);
}
```

This is intriguing, given the other object relocation proposals currently before WG21 ([P1029] *move* `= relocates`; [P1144] *Object relocation in terms of move plus destroy*), as object detachment and attachment could be the general mechanism by which object values are relocated in memory.

This proposal paper will not propose how move construction and assignment could be modified to use object detachment and attachment instead, however P1029 would be the current expected plan (i.e. new code explicitly opts in to 'true moves', existing code does not change its behaviour). If this paper is smiled upon by WG21, P1029 would be upgraded to incorporate the use of object detachment and attachment instead of its current proposed mechanism.

2.5 Existing practice

This section should list well-known examples of code that use the above. The committee exists to standardize existing practice; the committee is also swayed by performance improvements. Showing both will be an important component of this paper.

3 Proposed solution

This section should describe each proposed operation and function individually, show examples of how it should be used, and explain why its use is not UB.

3.1 Design constraints and rationale

3.2 Object detachment/attachment

3.2.1 Detachable types

3.2.2 Detach cast

3.2.3 Attach cast

3.2.4 Ensuring stores

3.2.5 Ensuring loads

4 Discussion

4.1 Alternatives

What about `std::as_bytes<>` and `std::as_writable_bytes<>`?

What about [P1382] *`volatile_load<T>` and `volatile_store<T>`*?

5 Proposed Wording

Defer to next paper revision

6 Acknowledgments

Thanks to Jens Gustedt, Martin Uecker and Paul McKenney for their feedback.

7 References

[N4455] *No Sane Compiler Would Optimize Atomics*
https://wg21.link/N4455

[N4810] *Working Draft, Standard for Programming Language C++*
https://wg21.link/N4810

[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

[P1026] Douglas, Niall
A call for a Data Persistence (`iostream` v2) study group
https://wg21.link/P1026

[P1029] Douglas, Niall
move `= relocates`
https://wg21.link/P1029

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

[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

[P1382] JF Bastien, Paul McKenney, Jeffrey Yasskin, et al
`volatile_load<T>` and `volatile_store<T>`
https://wg21.link/P1382

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