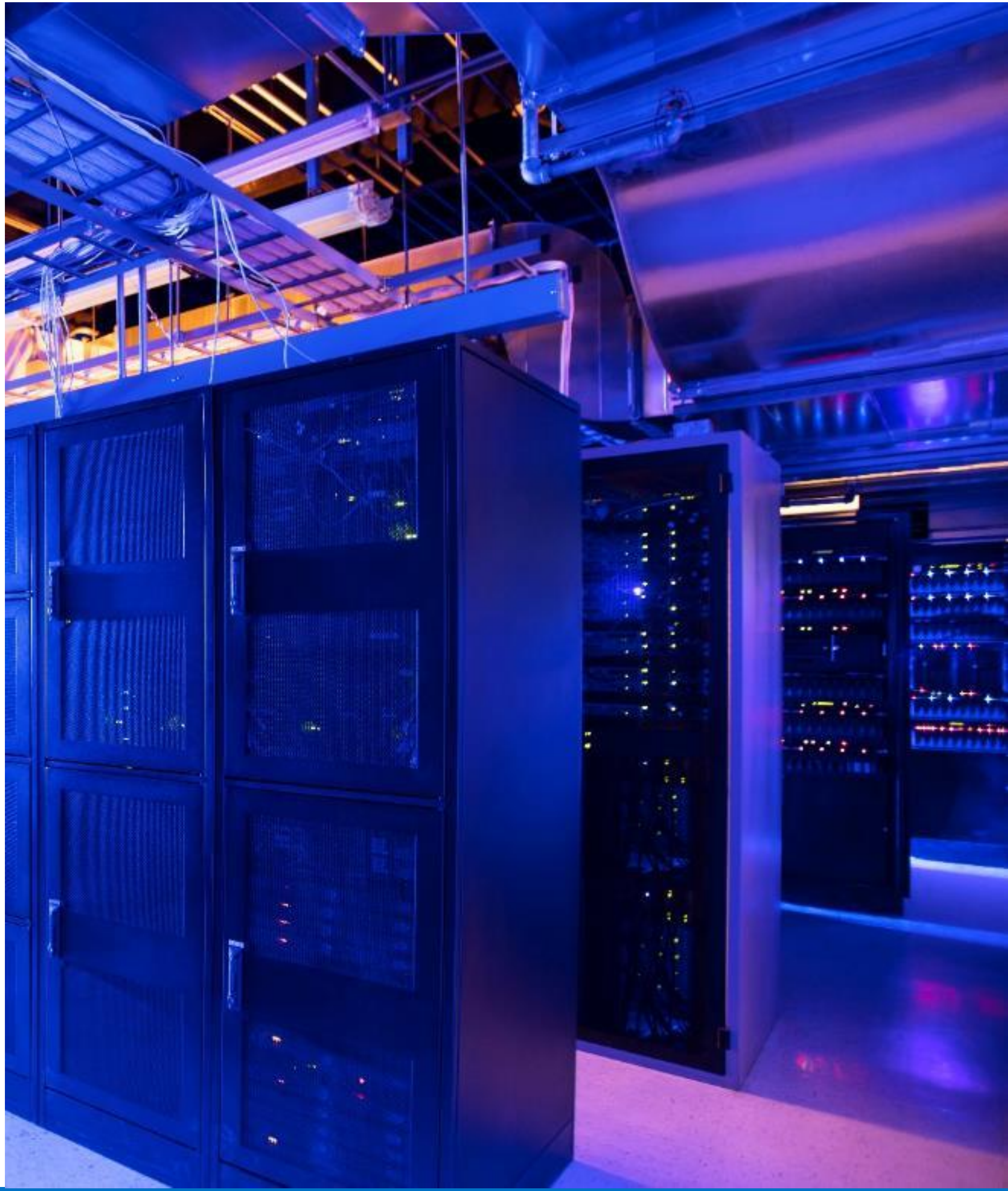
CREATING C++ APPS WITH LIBPMEMOBJ

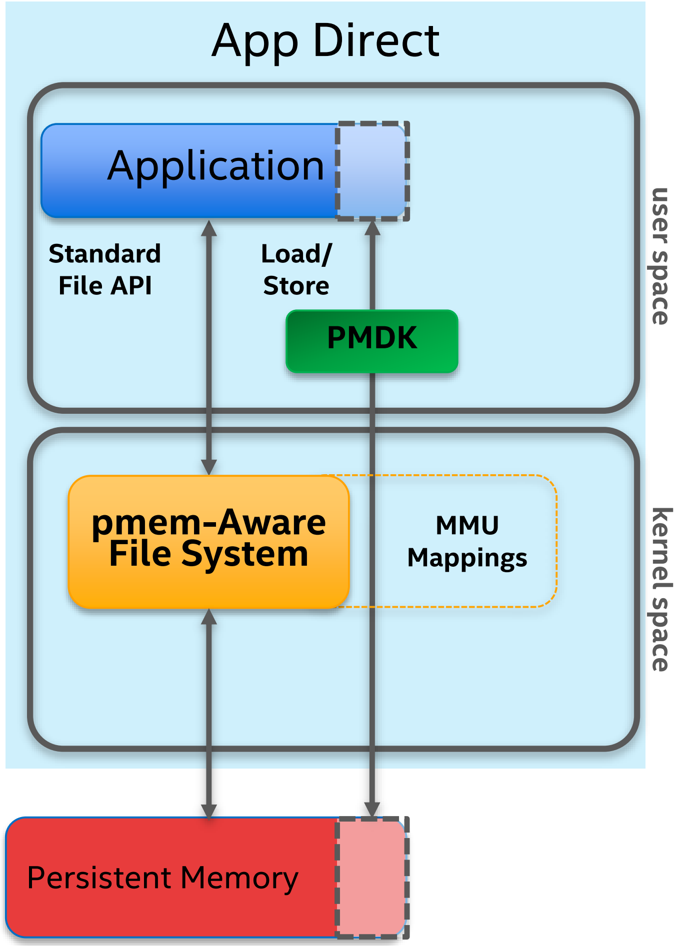
AGENDA

- App Direct mode
- PMDK and libpmemobj
- Persistent Memory pool
- Persistent pointer
- Root object
- Transactions
- `pmem::obj::p`
- Persistent Memory allocations
- Persistent Memory containers
- Example
- C++ standard limitations



APP DIRECT MODE

App Direct mode



Different modes for using Persistent Memory:

- Memory Mode
- Storage over App Direct
- App Direct

In-place persistence (no paging, context switching, interrupts, nor kernel code executes)

Byte addressable like memory (Load/store access, no page caching)

Cache Coherent

A pmem-aware file system exposes persistent memory to applications as files.

```
fd = open("/my/file", O_RDWR);
…
base = mmap(NULL, filesize,
                        PROT_READ|PROT_WRITE,
                        MAP_SHARED_VALIDATE|MAP_SYNC, fd, 0);
close(fd);
…
base[100] = 'X';
strcpy(base, "hello there");
msync(…);
…
```

Application must take responsibility for recovery, consistency and atomicity.

PMDK AND LIBPMEMOBJ

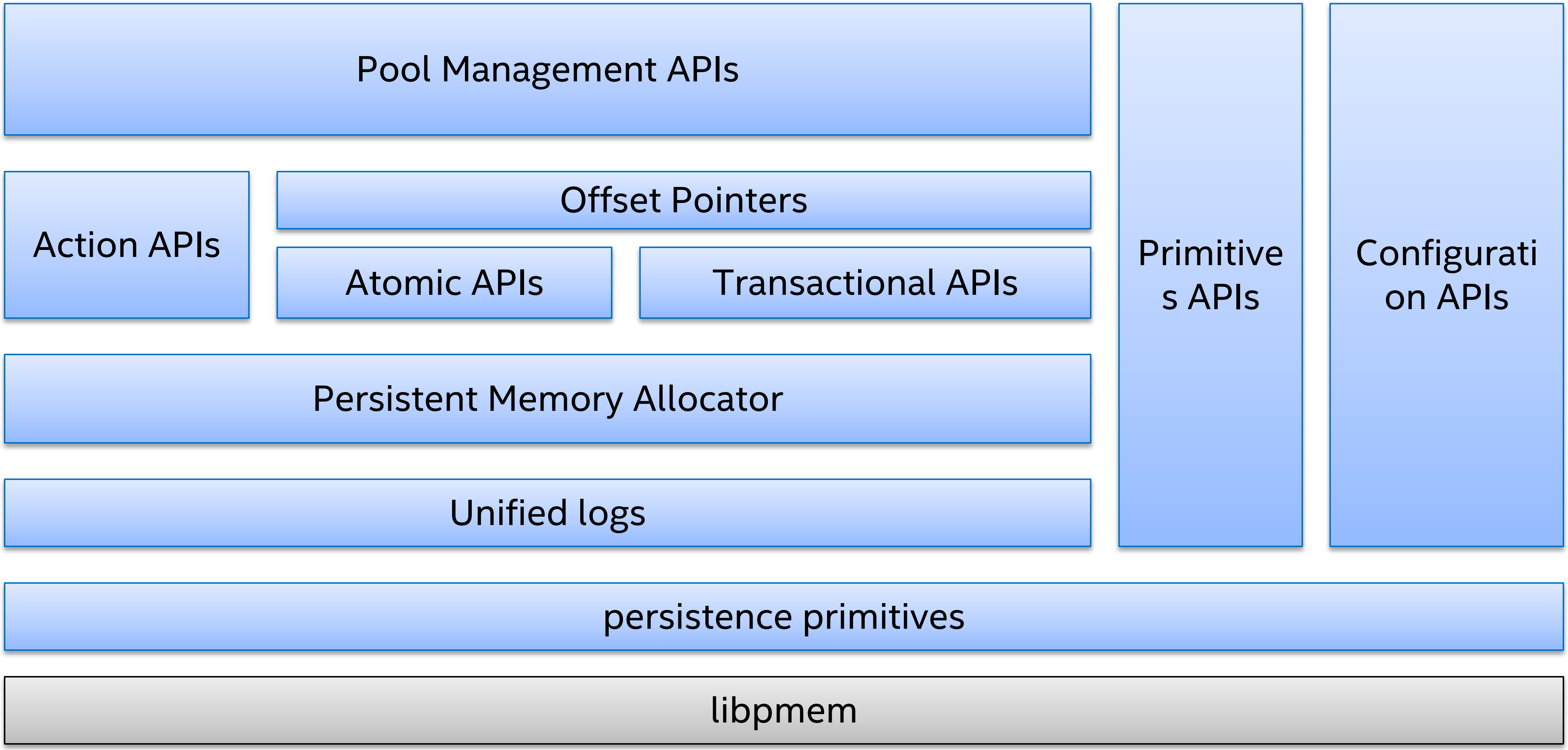
PMDK and libpmemobj

- http://pmem.io/
- open-source https://github.com/pmem
- vendor-agnostic
- user-space
- production quality, fully documented
- performance optimized and tuned

Software stack:

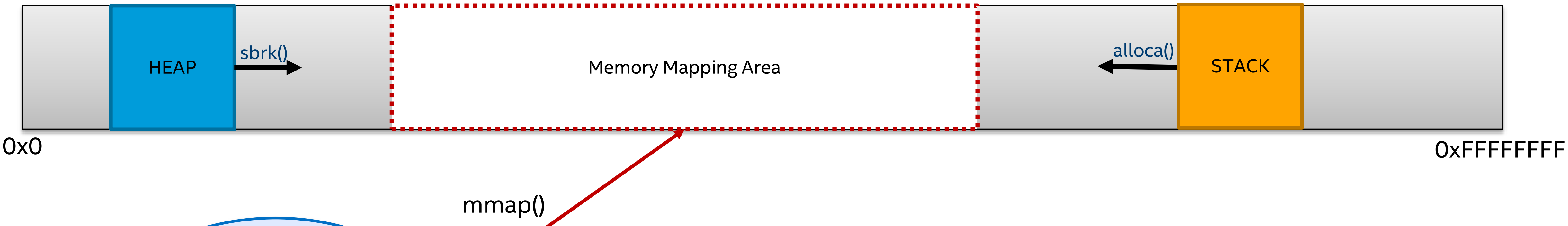
C++ idiomatic bindings for libpmemobj	**libpmemobj++**
Interface for persistent memory allocation, transactions and general facilities	**libpmemobj**
Low level support for local persistent memory	**libpmem**

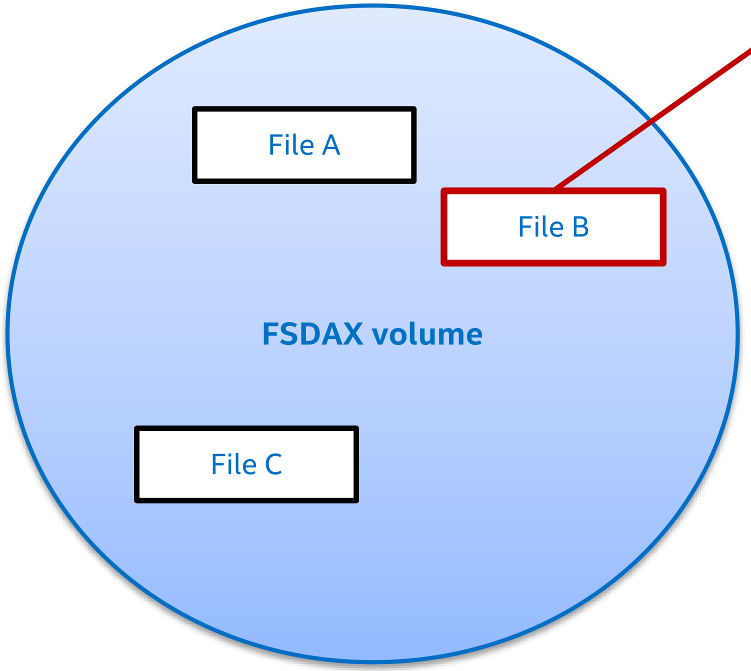
libpmemobj


Pool Management APIs
Action APIs
Offset Pointers
Atomic APIs
Transactional APIs
Primitives APIs
Configuration APIs
Persistent Memory Allocator
Unified logs
persistence primitives
libpmem

PERSISTENT MEMORY POOL

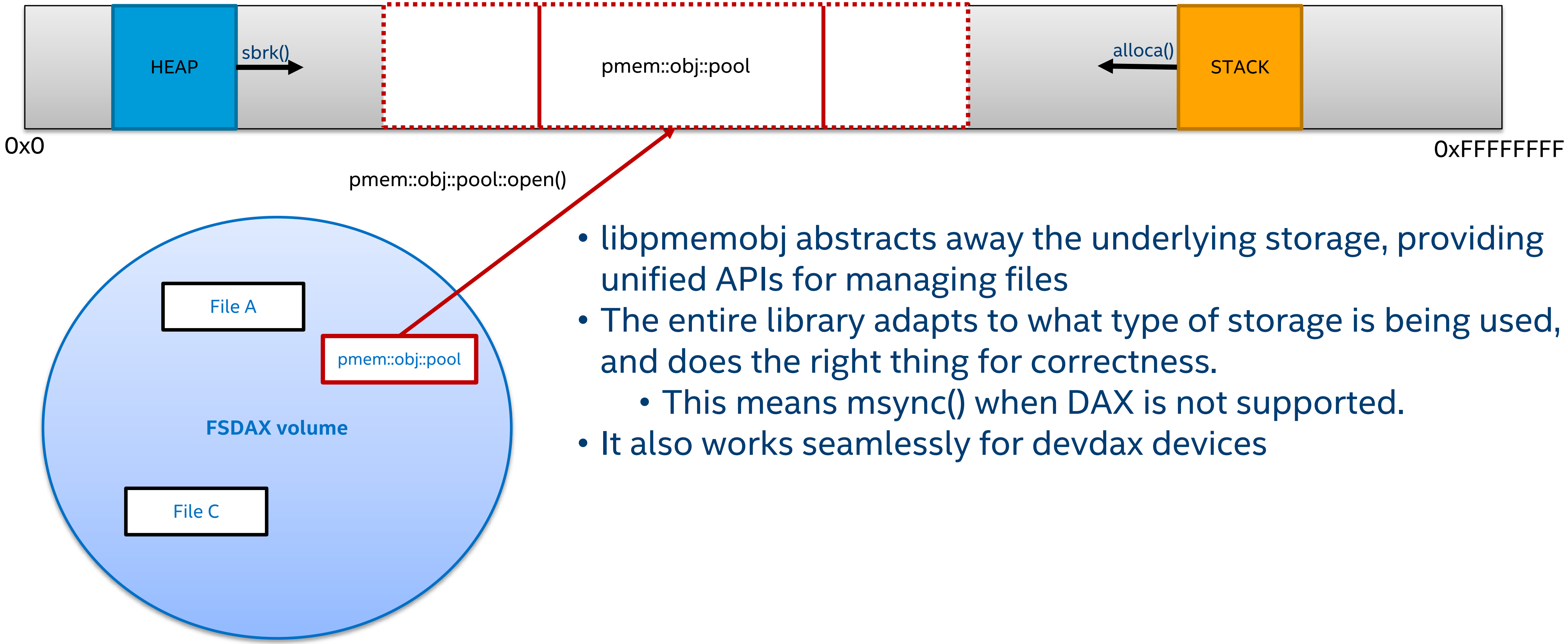
Pool Management APIs





- Persistent Memory is usually exposed by the OS through a DAX-enabled file system.
- Memory Mapping is used to take advantage of byte-addressability of PMEM
- mmap() does **not** guarantee the address of the mapping. Especially if Address Space Layout Randomization (ASLR) is enabled.

Pool Management APIs



- libpmemobj abstracts away the underlying storage, providing unified APIs for managing files
- The entire library adapts to what type of storage is being used, and does the right thing for correctness.
  - This means msync() when DAX is not supported.
- It also works seamlessly for devdax devices

http://pmem.io/libpmemobj-cpp/master/doxygen/classpmem_1_1obj_1_1pool.html

Pool Management APIs

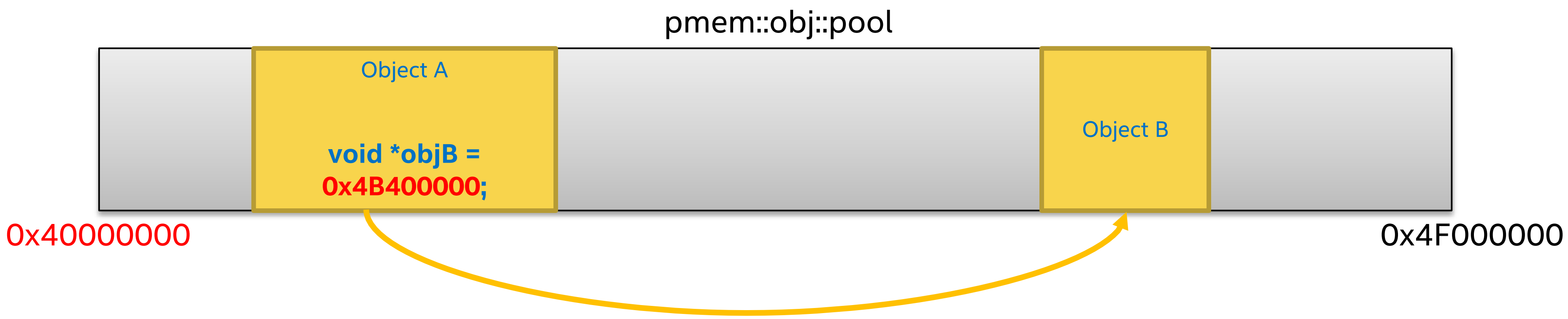
pool<> class example

```
if (access(path.c_str(), F_OK) != 0) {
    pop = pool<root>::create(path, "some_layout", PMEMOBJ_MIN_POOL, S_IRWXU);
} else {
    pop = pool<root>::open(path, "some_layout");
}
```

- Class template, where the template parameter is the type of the root object
- Supports basic operations
  - open – opens an existing pmempobj pool
  - create – creates a new pmemobj pool
  - close – closes an already opened/created pool
  - root – returns persistent pointer to root object associated with pool
- Inherits from pool_base

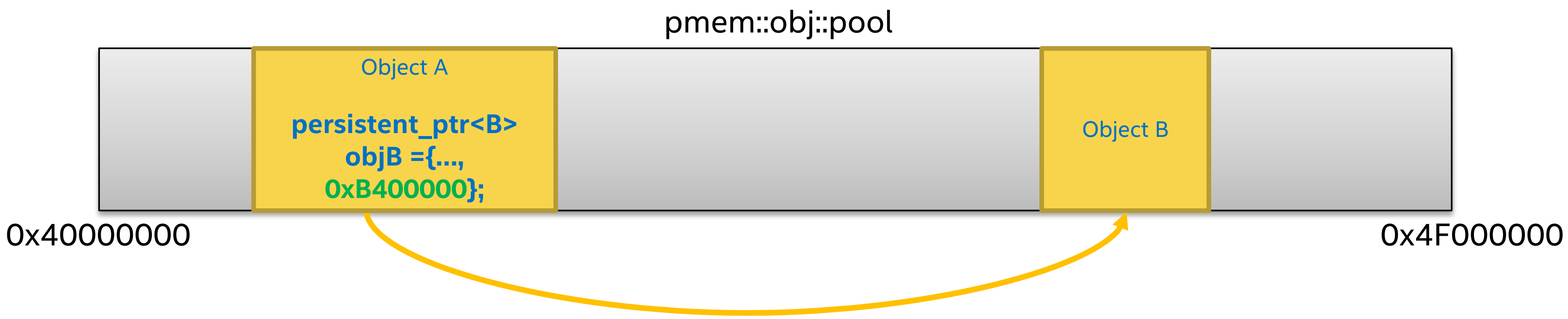
PERSISTENT POINTER

Persistent Pointer



- The base pointer of the mapping can change between application instances
- This means that any raw pointers between two memory locations can become invalid
- Must either fix all the pointers at the start of the application
    - Potentially terabytes of data to go through…
- Or use a custom data structure which isn't relative to the base pointer

Persistent Pointer

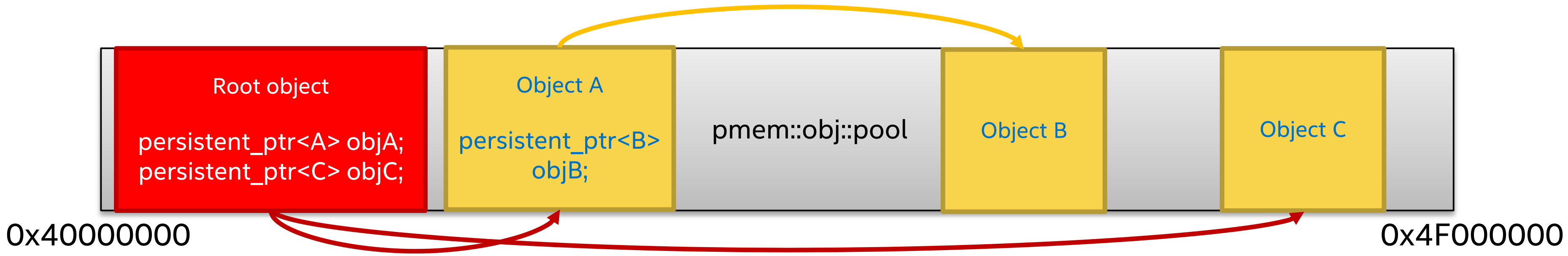


- libpmemobj provides 16 byte offset pointers, which contain an offset relative to the beginning of the mapping.
- Is a random access iterator
- Has primitives for flushing contents to persistence
- Does not manage object lifetime
- Does not automatically add contents to the transaction
- But it does add itself to the transaction

http://pmem.io/libpmemobj-cpp/master/doxygen/classpmem_1_1obj_1_1persistent__ptr.html

ROOT OBJECT

Root object



- All data structures of an application start at the root object.
- Has user-defined size, always exists and is initially zeroed.

- Applications should make sure that all objects are always reachable through some path that starts at the root object.
- Unreachable objects are effectively persistent memory leaks.

http://pmem.io/libpmemobj-cpp/master/doxygen/classpmem_1_1obj_1_1pool.html#a85b71b78e8229b009639835a5ad159d2

Root object

Retrieving root object from pool handle example

```
struct foo {
        persistent_ptr<bar> barp;
        long long x;
};

pop = pool<foo>::create(…);   // use “foo” type as a root

persistent_ptr<foo> r = pop.root();
assert(r->barp == nullptr);   // how to allocate an object of type “bar” in
                              // persistent memory?

r->x = 100;    // how to assign new value and guarantee data consistency?
               // What if crash happens during execution of this line?
```

TRANSACTIONS

Transactional API

- libpmemobj provides ACID (Atomicity, Consistency, Isolation, Durability) transactions for persistent memory
  - Atomicity means that a transaction either succeeds or fails completely
  - Consistency means that the transaction transforms `pmem::obj::pool` from one consistent state to another. This means that a pool won't get corrupted by a transaction.
  - Isolation means that transactions can be executed as if the operations were executed serially on the pool. This is optional, and requires user-provided locks.
  - Durability means that once a transaction is committed, it remains committed even in the case of system failures

Transactional API

Transaction example

```
auto pop = pool<root>::open("/path/to/poolfile", "layout string");

transaction::run(pop, [] {
    // do some work...
}, persistent_mtx, persistent_shmtx);
```

- Undo log based transactions
  - In case of interruption it is rolled-back or completed upon next pool open
- Take an `std::function` object as transaction body
- No explicit transaction commit
- Available with every C++11 compliant compiler
- Throw an exception when the transaction is aborted
- Take an arbitrary number of locks
- Can be nested

PMEM::OBJ::P

pmem::obj::p class

Code with manual snapshotting example

```
struct data {
    long long x;
}

auto pop = pool<data>::("/path/to/poolfile", "layout string");
auto datap = pop.root();

transaction::run(pop, [&]{
    pmemobj_tx_add_range(root, 0, sizeof (struct data));  // native C API
    datap->x = 5;
});
```

- If we won't snapshot data and the crash will occur during execution of transaction, the old value of "x" field won't be rolled-back

`pmem::obj::p` class

- Template class
- Overloads operator= for snapshotting in a transaction
- Overloads a bunch of other operators for seamless integration
  - Arithmetic
  - Logical
- Should be used for fundamental types
- No convenient way to access members of aggregate types
- No operator. to overload

pmem::obj::p class

Code with pmem::obj:p example

```
struct data {
    p<long long> x;
}

auto pop = pool<data>::("/path/to/poolfile", "layout string");
auto datap = pop.root();

transaction::run(pop, [&]{
    datap->x = 5;      // no need for implicit snapshotting
});
```

- More C++ idiomatic approach
- To modify your application and start using Persistent Memory, we should focus on modifying data structures, not functions

PERSISTENT MEMORY ALLOCATIONS

Persistent Memory allocations

- Can be used only within transactions
- Use transaction logic to enable allocation/delete rollback of persistent state
- make_persistent calls appropriate constructor
  - Syntax similar to std::make_shared
- delete_persistent calls the destructor
  - Not similar to anything found in std

Persistent Memory allocations

Transactional allocation example

```
struct data {
    data(p<int> a, p<int> b) : a(a), b(b) {}
    p<int> a;
    p<int> b;
}
transaction::run(pop, [&]{
    persistent_ptr<data> ptr = make_persistent<data>(1, 2);
    assert(ptr->a == 1);
    assert(ptr->b == 2);

    // more code here

    delete_persistent<data>(ptr);
});
```

PERSISTENT MEMORY CONTAINERS

Persistent Memory containers

- compatible interface with STL counterparts (almost)
- Takes care of adding elements to a transaction
  - In operator[]/at() when obtainig non-const reference
  - On iterator dereference
  - In other methods which allow write access to data
- Works with std algorithms
- All functions which may alter container properties are atomic
  - This includes: resize(), reserve(), push_back() and others
  - Transactions are used internally
  - Strong exception guarantee
- Currently (libpmemobj++ 1.7) available containers:
  - array
  - vector
  - string (implemented basisc operations)
  - concurrent_hash_map (no STL counterpart, used as an engine for pmemkv)

Persistent Memory containers

vector usage example

```
transaction::run(pop, [&] {
        root->vec_p = make_persistent<vector<int>>();
});

vector_type &pvector = *(root->vec_p);

pvector.resize(10);
pvector = {5, 4, 3, 2, 1};
pvector.push_back(0);

transaction::run(pop, [&]{
    std::sort(pvector.begin(), pvector.end()); // 0,1,2,3,4,5

    delete_persistent<vector<int>>(ptr);
});
```

EXAMPLE

Example

volatile queue -> persistent queue

```
struct queue_node {
    int value;
    struct queue_node *next;
};

struct queue {
    …
    void
    push(int value)
    {
        auto node = new queue_node;
        node->value = value;
        node->next = nullptr;

        if (head == nullptr) {
            head = tail = node;
        } else {
            tail->next = node;
            tail = node;
        }
    }
```

```
struct queue_node {
    p<int> value;
    persistent_ptr<struct queue_node> next;
};

struct queue {
    …
    void
    push(pool_base &pop, int value)
    {
        transaction::run(pop, [&] {
        auto node = make_persistent<queue_node>();
        node->value = value;
        node->next = nullptr;

        if (head == nullptr) {
            head = tail = node;
        } else {
            tail->next = node;
            tail = node;
        }
        });
    }
```

Example

volatile queue -> persistent queue

```
int
pop() {
    if (head == nullptr)
        throw std::out_of_range("no elements");

    auto head_ptr = head;
    auto value = head->value;

    head = head->next;
    delete head_ptr;

    if (head == nullptr)
        tail = nullptr;

    return value;
}

…
```

```
int
pop(pool_base &pop) {
    int value;
    transaction::run(pop, [&] {
    if (head == nullptr)
        throw std::out_of_range("no elements");

    auto head_ptr = head;
    value = head->value;

    head = head->next;
    delete_persistent<queue_node>(head_ptr);

    if (head == nullptr)
        tail = nullptr;
    });

    return value;
}

…
```

C++ STANDARD LIMITATIONS

C++ standard limitations

- Object lifetime begins when initialization is completed (constructor is called) and end when destructor calls starts
  - Similar problem to transmitting data over network (where the C++ application is given an array of bytes but might be able to recognize the type of object sent)
  - problem is well known and is being addressed by WG21 (The C++ Standards Committee Working Group)
  - For now, we must rely on undefined behavior
- Snapshotting – data is being copied with `memcpy()` and it means that we may break the inherent behavior of the object which may rely on the copy constructor
  - `std::is_trivially_copyable` should guarantee safe copying raw bytes, but is a restrictive type-trait (no user provided copy/move constructors)

C++ standard limitations

- Object layout:
  - might differ between compilers/compiler flags/ABI
  - compiler may do some layout-related optimizations and is free to shuffle order of members with same specifier type (public/protected/public)
  - No polimorfic types are allowed: there is no reliable and portable way to implement vtable rebuilding after reopening the pool

```
someType A = *reinterpret_cast<someType*>(mmap(...));
```

  - the bit representation of stored object type must be always the same and application should be able to retrieve stored object from memory mapped file without serialization.
  - `std::is_standard_layout` guarantee fixed layout, but is very restrictive type-trait

C++ standard limitations

- Storing volatile memory pointers in persistent memory is almost always a design error

```
class rootType {
        int* vptr;
}

...

int val = 1; /* variable on stack */
pmem::obj::transaction::run(pop, [&](){
        root->vptr = &val;
};);
```

- Using `pmem::obj::persistent_ptr<>` class template is safe, and it provides only way to access specific memory area after application crash
  - It doesn't satisfy requirements of `std::is_trivially_copyable` check
  - We rely on undefined behavior
- Type restrictions should not be a problem for native Persistent Memory applications – to fully utilize PMEM advantages, user should consider data oriented design principles

experience
what's inside™