CHAPTER 8

libpmemobj-cpp: The Adaptable Language - C++ and Persistent Memory

Introduction

The Persistent Memory Development Kit (PMDK) includes several separate libraries; each is designed with a specific use in mind. The most flexible and powerful one is `libpmemobj`. It complies with the persistent memory programming model without modifying the compiler. Intended for developers of low-level system software and language creators, the `libpmemobj` library provides allocators, transactions, and a way to automatically manipulate objects. Because it does not modify the compiler, its API is verbose and macro heavy.

To make persistent memory programming easier and less error prone, higher-level language bindings for `libpmemobj` were created and included in PMDK. The C++ language was chosen to create new and friendly API to `libpmemobj` called `libpmemobj-cpp`, which is also referred to as `libpmemobj++`. C++ is versatile, feature rich, has a large developer base, and it is constantly being improved with updates to the C++ programming standard.

The main goal for the `libpmemobj-cpp` bindings design was to focus modifications to volatile programs on data structures and not on the code. In other words, `libpmemobj-cpp` bindings are for developers, who want to modify volatile applications, provided with a convenient API for modifying structures and classes with only slight modifications to functions.

S. Scargall, *Programming Persistent Memory*, https://doi.org/10.1007/978-1-4842-4932-1_8

This chapter describes how to leverage the C++ language features that support metaprogramming to make persistent memory programming easier. It also describes how to make it more C++ idiomatic by providing persistent containers. Finally, we discuss C++ standard limitations for persistent memory programming, including an object's lifetime and the internal layout of objects stored in persistent memory.

Metaprogramming to the Rescue

Metaprogramming is a technique in which computer programs have the ability to treat other programs as their data. It means that a program can be designed to read, generate, analyze or transform other programs, and even modify itself while running. In some cases, this allows programmers to minimize the number of lines of code to express a solution, in turn reducing development time. It also allows programs greater flexibility to efficiently handle new situations without recompilation.

For the `libpmemobj-cpp` library, considerable effort was put into encapsulating the PMEMoids (persistent memory object IDs) with a type-safe container. Instead of a sophisticated set of macros for providing type safety, templates and metaprogramming are used. This significantly simplifies the native C `libpmemobj` API.

Persistent Pointers

The persistent memory programming model created by the Storage Networking Industry Association (SNIA) is based on memory-mapped files. PMDK uses this model for its architecture and design implementation. We discussed the SNIA programming model in Chapter 3.

Most operating systems implement address space layout randomization (ASLR). ASLR is a computer security technique involved in preventing exploitation of memory corruption vulnerabilities. To prevent an attacker from reliably jumping to, for example, a particular exploited function in memory, ASLR randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap, and libraries. Because of ASLR, files can be mapped at different addresses of the process address space each time the application executes. As a result, traditional pointers that store absolute addresses cannot be used. Upon each execution, a traditional pointer might point to uninitialized memory for which

dereferencing it may result in a segmentation fault. Or it might point to a valid memory range, but not the one that the user expects it to point to, resulting in unexpected and undetermined behavior.

To solve this problem in persistent memory programming, a different type of pointer is needed. `libpmemobj` introduced a C struct called `PMEMoid`, which consists of an identifier of the pool and an offset from its beginning. This *fat pointer* is encapsulated in `libpmemobj` C++ bindings as a template class `pmem::obj::persistent_ptr.` Both the C and C++ implementations have the same 16-byte footprint. A constructor from raw `PMEMoid` is provided so that mixing the C API with C++ is possible. The `pmem::obj::persistent_ptr` is similar in concept and implementation to the smart pointers introduced in C++11 (`std::shared_ptr`, `std::auto_ptr`, `std::unique_ptr`, and `std::weak_ptr`), with one big difference – it does not manage the object's life cycle.

Besides `operator*`, `operator->`, `operator[]`, and `typedefs` for compatibility with `std::pointer_traits` and `std::iterator_traits`, the `pmem::obj::persistent_ptr` also has defined methods for persisting its contents. The `pmem::obj::persistent_ptr` can be used in standard library algorithms and containers.

Transactions

Being able to modify more than 8 bytes of storage at a time atomically is imperative for most nontrivial algorithms one might want to use in persistent memory. Commonly, a single logical operation requires multiple stores. For example, an insert into a simple list-based queue requires two separate stores: a tail pointer and the next pointer of the last element. To enable developers to modify larger amounts of data atomically, with respect to power-fail interruptions, the PMDK library provides transaction support in some of its libraries. The C++ language bindings wrap these transactions into two concepts: one, based on the resource acquisition is initialization (RAII) idiom and the other based on a callable `std::function` object. Additionally, because of some C++ standard issues, the scoped transactions come in two flavors: manual and automatic. In this chapter we only describe the approach with `std::function` object. For information about RAII-based transactions, refer to `libpmemobj-cpp` documentation (https://pmem.io/pmdk/cpp_obj/).

The method which uses `std::function` is declared as

```
void pmem::obj::transaction::run(pool_base &pop,
    std::function<void ()> tx, Locks&... locks)
```

The `locks` parameter is a variadic template. Thanks to the `std::function`, a myriad of types can be passed in to `run`. One of the preferred ways is to pass a lambda function as the `tx` parameter. This makes the code compact and easier to analyze. Listing 8-1 shows how lambda can be used to perform work in a transaction.

Listing 8-1. Function object transaction

```
45        // execute a transaction
46        pmem::obj::transaction::run(pop, [&]() {
47            // do transactional work
48        });
```

Of course, this API is not limited to just lambda functions. Any callable target can be passed as `tx`, such as functions, bind expressions, function objects, and pointers to member functions. Since `run` is a normal static member function, it has the benefit of being able to throw exceptions. If an exception is thrown during the execution of a transaction, it is automatically aborted, and the active exception is rethrown so information about the interruption is not lost. If the underlying C library fails for any reason, the transaction is also aborted, and a C++ library exception is thrown. The developer is no longer burdened with the task of checking the status of the previous transaction.

`libpmemobj-cpp` transactions provide an entry point for persistent memory resident synchronization primitives such as `pmem::obj::mutex`, `pmem::obj::shared_mutex` and `pmem::obj::timed_mutex`. `libpmemobj` ensures that all locks are properly reinitialized when one attempts to acquire a lock for the first time. The use of pmem locks is completely optional, and transactions can be executed without them. The number of supplied locks is arbitrary, and the types can be freely mixed. The locks are held until the end of the given transaction, or the outermost transaction in the case of nesting. This means when transactions are enclosed by a try-catch statement, the locks are released before reaching the catch clause. This is extremely important in case some kind of transaction abort cleanup needs to modify the shared state. In such a case, the necessary locks need to be reacquired in the correct order.

Snapshotting

The C library requires manual snapshots before modifying data in a transaction. The C++ bindings do all of the snapshotting automatically, to reduce the probability of programmer error. The `pmem::obj::p` template wrapper class is the basic building block for this mechanism. It is designed to work with basic types and not compound types such as classes or PODs (*Plain Old Data*, structures with fields only and without any object-oriented features). This is because it does not define `operator->()` and there is no possibility to implement `operator.()`. The implementation of `pmem::obj::p` is based on the `operator=()`. Each time the assignment operator is called, the value wrapped by `p` will be changed, and the library needs to snapshot the old value. In addition to snapshotting, the `p<>` template ensures the variable is persisted correctly, flushing data if necessary. Listing 8-2 provides an example of using the `p<>` template.

Listing 8-2. Using the p<> template to persist values correctly

```
39  struct bad_example {
40      int some_int;
41      float some_float;
42  };
43
44  struct good_example {
45      pmem::obj::p<int> pint;
46      pmem::obj::p<float> pfloat;
47  };
48
49  struct root {
50      bad_example bad;
51      good_example good;
52  };
53
54  int main(int argc, char *argv[]) {
55      auto pop = pmem::obj::pool<root>::open("/daxfs/file", "p");
56
57      auto r = pop.root();
58
```

```
59        pmem::obj::transaction::run(pop, [&]() {
60            r->bad.some_int = 10;
61            r->good.pint = 10;
62
63            r->good.pint += 1;
64        });
65
66        return 0;
67    }
```

- Lines 39-42: Here, we declare a `bad_example` structure with two variables – `some_int` and `some_float`. Storing this structure on persistent memory and modifying it are dangerous because data is not snapshotted automatically.
- Lines 44-47: We declare the `good_example` structure with two `p<>` type variables – `pint` and `pfloat`. This structure can be safely stored on persistent memory as every modification of `pint` or `pfloat` in a transaction will perform a snapshot.
- Lines 55-57: Here, we open a persistent memory pool, created already using the pmempool command, and obtain a pointer to the root object stored within the `root` variable.
- Line 60: We modify the integer value from the `bad_example` structure. This modification is not safe because we do not add this variable to the transaction; hence it will not be correctly made persistent if there is an unexpected application or system crash or power failure.
- Line 61: Here, we modify integer value wrapped by p<> template. This is safe because `operator=()` will automatically snapshot the element.
- Line 63: Using arithmetic operators on `p<>` (if the underlying type supports it) is also safe.

Allocating

As with `std::shared_ptr`, the `pmem::obj::persistent_ptr` comes with a set of allocating and deallocating functions. This helps allocate memory and create objects, as well as destroy and deallocate the memory. This is especially important in the case of persistent

memory because all allocations and object construction/destruction must be done atomically with respect to power-fail interruptions. The transactional allocations use perfect forwarding and variadic templates for object construction. This makes object creation similar to calling the constructor and identical to `std::make_shared`. The transactional array creation, however, requires the objects to be default constructible. The created arrays can be multidimensional. The `pmem::obj::make_persistent` and `pmem::obj::make_persistent_array` must be called within a transaction; otherwise, an exception is thrown. During object construction, other transactional allocations can be made, and that is what makes this API very flexible. The specifics of persistent memory required the introduction of the `pmem::obj::delete_persistent` function, which destroys objects and arrays of objects. Since the `pmem::obj::persistent_ptr` does not automatically handle the lifetime of pointed to objects, the user is responsible for disposing of the ones that are no longer in use. Listing 8-3 shows example of transaction allocation.

Atomic allocations behave differently as they do not return a pointer. Developers must provide a reference to one as the function's argument. Because atomic allocations are not executed in the context of a transaction, the actual pointer assignment must be done through other means. For example, by redo logging the operation. Listing 8-3 also provides an example of atomic allocation.

Listing 8-3. Example of transactional and atomic allocations

```
39    struct my_data {
40        my_data(int a, int b): a(a), b(b) {
41
42        }
43
44        int a;
45        int b;
46    };
47
48    struct root {
49        pmem::obj::persistent_ptr<my_data> mdata;
50    };
51
52    int main(int argc, char *argv[]) {
53        auto pop = pmem::obj::pool<root>::open("/daxfs/file", "tx");
```

```
54
55          auto r = pop.root();
56
57          pmem::obj::transaction::run(pop, [&]() {
58              r->mdata = pmem::obj::make_persistent<my_data>(1, 2);
59          });
60
61          pmem::obj::transaction::run(pop, [&]() {
62              pmem::obj::delete_persistent<my_data>(r->mdata);
63          });
64          pmem::obj::make_persistent_atomic<my_data>(pop, r->mdata,
            2, 3);
65
66          return 0;
67      }
```

- Line 58: Here, we allocate my_data object transactionally. Parameters passed to `make_persistent` will be forwarded to `my_data` constructor. Note that assignment to `r->mdata` will perform a snapshot of old persistent pointer's value.
- Line 62: Here, we delete the `my_data` object. `delete_persistent` will call the object's destructor and free the memory.
- Line 64: We allocate `my_data` object atomically. Calling this function **cannot** be done inside of a transaction.

C++ Standard limitations

The C++ language restrictions and persistent memory programming paradigm imply serious restrictions on objects which may be stored on persistent memory. Applications can access persistent memory with memory-mapped files to take advantage of its byte addressability thanks to `libpmemobj` and SNIA programming model. No serialization takes place here, so applications must be able to read and modify directly from the persistent memory media even after the application was closed and reopened or after a power failure event.

What does the preceding mean from a C++ and libpmemobj's perspective? There are four major problems:

1. Object lifetime
2. Snapshotting objects in transactions
3. Fixed on-media layout of stored objects
4. Pointers as object members

These four problems will be described in next four sections.

An Object's Lifetime

The lifetime of an object is described in the [basic.life] section of the C++ standard (https://isocpp.org/std/the-standard):

> *The lifetime of an object or reference is a runtime property of the object or reference. A variable is said to have vacuous initialization if it is default-initialized and, if it is of class type or a (possibly multi-dimensional) array thereof, that class type has a trivial default constructor. The lifetime of an object of type T begins when:*
>
> *(1.1) storage with the proper alignment and size for type T is obtained, and*
>
> *(1.2) its initialization (if any) is complete (including vacuous initialization) ([dcl.init]), 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 ([dcl.init.aggr], [class.base.init]), or as described in [class.union]. The lifetime of an object of type T ends when:*
>
> *(1.3) if T is a non-class type, the object is destroyed, or*
>
> *(1.4) if T is a class type, the destructor call starts, or*
>
> *(1.5) the storage which the object occupies is released, or is reused by an object that is not nested within o ([intro.object]).*

The standard states that properties ascribed to objects apply for a given object only during its lifetime. In this context, the persistent memory programming problem is similar to transmitting data over a network, where the C++ application is given an array of bytes but might be able to recognize the type of object sent. However, the object was not constructed in this application, so using it would result in undefined behavior.

This problem is well known and is being addressed by the WG21 C++ Standards Committee Working Group (https://isocpp.org/std/the-committee and http://www.open-std.org/jtc1/sc22/wg21/).

Currently, there is no possible way to overcome the object-lifetime obstacle and stop relying on undefined behavior from C++ standard's point of view. `libpmemobj-cpp` is tested and validated with various C++11 compliant compilers and use case scenarios. The only recommendation for `libpmemobj-cpp` users is that they must keep this limitation in mind when developing persistent memory applications.

Trivial Types

Transactions are the heart of `libpmemobj`. That is why `libpmemobj-cpp` was implemented with utmost care while designing the C++ versions so they are as easy to use as possible. Developers do not have to know the implementation details and do not have to worry about snapshotting modified data to make undo log–based transaction works. A special semi-transparent template property class has been implemented to automatically add variable modifications to the transaction undo log, which is described in the "Snapshotting" section.

But what does snapshotting data mean? The answer is very simple, but the consequences for C++ are not. `libpmemobj` implements snapshotting by copying data of given length from a specified address to another address using `memcpy()`. If a transaction aborts or a system power loss occurs, the data will be written from the undo log when the memory pool is reopened. Consider a definition of the following C++ object, presented in Listing 8-4, and think about the consequences that a `memcpy()` has on it.

Listing 8-4. An example showing an unsafe memcpy() on an object

```
35    class nonTriviallyCopyable {
36    private:
37        int* i;
38    public:
39        nonTriviallyCopyable (const nonTriviallyCopyable & from)
40        {
41            /* perform non-trivial copying routine */
42            i = new int(*from.i);
43        }
44    };
```

Deep and shallow copying is the simplest example. The gist of the problem is that by copying the data manually, we may break the inherent behavior of the object which may rely on the copy constructor. Any shared or unique pointer would be another great example – by simple copying it with `memcpy()`, we break the "deal" we made with that class when we used it, and it may lead to leaks or crashes.

The application must handle many more sophisticated details when it manually copies the contents of an object. The C++11 standard provides a `<type_traits>` type trait and `std::is_trivially_copyable`, which ensure a given type satisfies the requirements of *TriviallyCopyable*. Referring to C++ standard, an object satisfies the *TriviallyCopyable* requirements when

> *A trivially copyable class is a class that:*
>
> *— has no non-trivial copy constructors (12.8),*
>
> *— has no non-trivial move constructors (12.8),*
>
> *— has no non-trivial copy assignment operators (13.5.3, 12.8),*
>
> *— has no non-trivial move assignment operators (13.5.3, 12.8), and*
>
> *— has a trivial destructor (12.4).*
>
> *A trivial class is a class that has a trivial default constructor (12.1) and is trivially copyable.*
>
> *[Note: In particular, a trivially copyable or trivial class does not have virtual functions or virtual base classes.]*

The C++ standard defines nontrivial methods as follows:

> *A copy/move constructor for class X is trivial if it is not user-provided and if*
>
> *— class X has no virtual functions (10.3) and no virtual base classes (10.1), and*
>
> *— the constructor selected to copy/move each direct base class subobject is trivial, and*
>
> *— for each non-static data member of X that is of class type (or array thereof), the constructor selected to copy/move that member is trivial;*
>
> *otherwise, the copy/move constructor is non-trivial.*

This means that a copy or move constructor is trivial if it is not user provided. The class has nothing virtual in it, and this property holds recursively for all the members of the class and for the base class. As you can see, the C++ standard and `libpmemobj` transaction implementation limit the possible objects type to store on persistent memory to satisfy requirements of trivial types, but the layout of our objects must be taken into account.

Object Layout

Object representation, also referred to as the *layout*, might differ between compilers, compiler flags, and application binary interface (ABI). The compiler may do some layout-related optimizations and is free to shuffle order of members with same specifier type – for example, `public` then `protected`, then `public` again. Another problem related to unknown object layout is connected to polymorphic types. Currently there is no reliable and portable way to implement `vtable` rebuilding after reopening the memory pool, so polymorphic objects cannot be supported with persistent memory.

If we want to store objects on persistent memory using memory-mapped files and to follow the SNIA NVM programming model, we must ensure that the following casting will be always valid:

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

The bit representation of a stored object type must be always the same, and our application should be able to retrieve the stored object from the memory-mapped file without serialization.

It is possible to ensure that specific types satisfy the aforementioned requirements. C++11 provides another type trait called `std::is_standard_layout`. The standard mentions that it is useful for communicating with other languages, such as for creating language bindings to native C++ libraries as an example, and that's why a standard-layout class has the same memory layout of the equivalent C `struct` or `union`. A general rule is that standard-layout classes must have all non-static data members with the same access control. We mentioned this at the beginning of this section – that a C++ compliant compiler is free to shuffle access ranges of the same class definition.

When using inheritance, only one class in the whole inheritance tree can have non-static data members, and the first non-static data member cannot be of a base class type because this could break aliasing rules. Otherwise, it is not a standard-layout class.

The C++11 standard defines `std::is_standard_layout` as follows:

> *A standard-layout class is a class that:*
>
> *— has no non-static data members of type non-standard-layout class (or array of such types) or reference,*
>
> *— has no virtual functions (10.3) and no virtual base classes (10.1),*
>
> *— has the same access control (Clause 11) for all non-static data members,*
>
> *— has no non-standard-layout base classes,*
>
> *— either has no non-static data members in the most derived class and at most one base class with non-static data members, or has no base classes with non-static data members, and*
>
> *— has no base classes of the same type as the first non-static data member.*
>
> *A standard-layout struct is a standard-layout class defined with the class-key struct or the class-key class.*
>
> *A standard-layout union is a standard-layout class defined with the class-key union.*
>
> *[Note: Standard-layout classes are useful for communicating with code written in other programming languages. Their layout is specified in 9.2.]*

Having discussed object layouts, we look at another interesting problem with pointer types and how to store them on persistent memory.

Pointers

In previous sections, we quoted parts of the C++ standard. We were describing the limits of types which were safe to snapshot and copy and which we can binary-cast without thinking of fixed layout. But what about pointers? How do we deal with them in our objects as we come to grips with the persistent memory programming model? Consider the code snippet presented in Listing 8-5 which provides an example of a class that uses a volatile pointer as a class member.

Listing 8-5. Example of class with a volatile pointer as a class member

```
39  struct root {
40      int* vptr1;
41      int* vptr2;
42  };
43
44  int main(int argc, char *argv[]) {
45      auto pop = pmem::obj::pool<root>::open("/daxfs/file", "tx");
46
47      auto r = pop.root();
48
49      int a1 = 1;
50
51      pmem::obj::transaction::run(pop, [&](){
52          auto ptr = pmem::obj::make_persistent<int>(0);
53          r->vptr1 = ptr.get();
54          r->vptr2 = &a1;
55      });
56
57      return 0;
58  }
```

- Lines 39-42: We create a `root` structure with two volatile pointers as members.
- Lines 51-52: Our application is assigning, transactionally, two virtual addresses. One to an integer residing on the stack and the second to an integer residing on persistent memory. What will happen if the application crashes or exits after execution of the transaction and we execute the application again? Since the variable `a1` was residing on the stack, the old value vanished. But what is the value assigned to `vptr1`? Even if it resides on persistent memory, the volatile pointer is no longer valid. With ASLR we are not guaranteed to get the same virtual address again if we call `mmap()`. The pointer could point to something, nothing, or garbage.

As shown in the preceding example, it is very important to realize that storing volatile memory pointers in persistent memory is almost always a design error. However, using the `pmem::obj::persistent_ptr<>` class template is safe. It provides the only way to safely access specific memory after an application crash. However, the `pmem::obj::persistent_ptr<>` type does not satisfy `TriviallyCopyable` requirements because of explicitly defined constructors. As a result, an object with a `pmem::obj::persistent_ptr<>` member will not pass the `std::is_trivially_copyable` verification check. Every persistent memory developer should always check whether `pmem::obj::persistent_ptr<>` could be copied in that specific case and that it will not cause errors and persistent memory leaks. Developers should realize that `std::is_trivially_copyable` is a syntax check only and it does not test the semantics. Using `pmem::obj::persistent_ptr<>` in this context leads to undefined behavior. There is no single solution to the problem. At the time of writing this book, the C++ standard does not yet fully support persistent memory programming, so developers must ensure that copying `pmem::obj::persistent_ptr<>` is safe to use in each case.

Limitations Summary

C++11 provides several very useful type traits for persistent memory programming. These are

- `template <typename T>`

  `struct std::is_pod;`
- `template <typename T>`

  `struct std::is_trivial;`
- `template <typename T>`

  `struct std::is_trivially_copyable;`
- `template <typename T>`

  `struct std::is_standard_layout;`

They are correlated with each other. The most general and restrictive is the definition of a POD type shown in Figure 8-1.

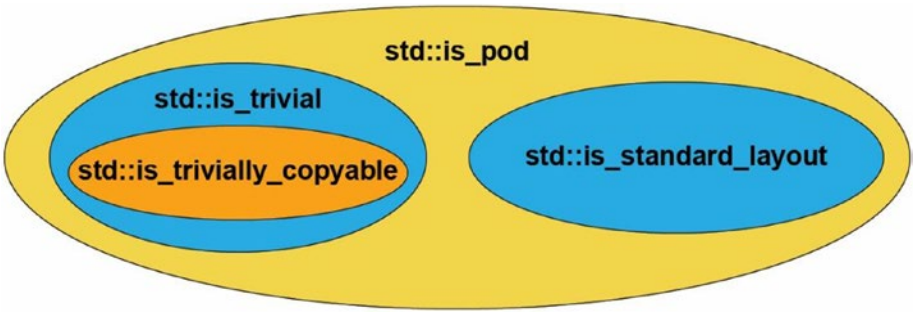

Figure 8-1. *Correlation between persistent memory–related C++ type traits*

We mentioned previously that a persistent memory resident class must satisfy the following requirements:

- `std::is_trivially_copyable`
- `std::is_standard_layout`

Persistent memory developers are free to use more restrictive type traits if required. If we want to use persistent pointers, however, we cannot rely on type traits, and we must be aware of all problems related to copying objects with `memcpy()` and the layout representation of objects. For persistent memory programming, a format description or standardization of the aforementioned concepts and features needs to take place within the C++ standards body group such that it can be officially designed and implemented. Until then, developers must be aware of the restrictions and limitations to manage undefined object-lifetime behavior.

Persistence Simplified

Consider a simple queue implementation, presented in Listing 8-6, which stores elements in volatile DRAM.

Listing 8-6. An implementation of a volatile queue

```
33    #include <cstdio>
34    #include <cstdlib>
35    #include <iostream>
36    #include <string>
37
```

```
38  struct queue_node {
39      int value;
40      struct queue_node *next;
41  };
42
43  struct queue {
44      void
45      push(int value)
46      {
47          auto node = new queue_node;
48          node->value = value;
49          node->next = nullptr;
50
51          if (head == nullptr) {
52              head = tail = node;
53          } else {
54              tail->next = node;
55              tail = node;
56          }
57      }
58
59      int
60      pop()
61      {
62          if (head == nullptr)
63              throw std::out_of_range("no elements");
64
65          auto head_ptr = head;
66          auto value = head->value;
67
68          head = head->next;
69          delete head_ptr;
70
71          if (head == nullptr)
72              tail = nullptr;
73
```

```
74              return value;
75          }
76
77          void
78          show()
79          {
80              auto node = head;
81              while (node != nullptr) {
82                  std::cout << "show: " << node->value << std::endl;
83                  node = node->next;
84              }
85
86              std::cout << std::endl;
87          }
88
89      private:
90          queue_node *head = nullptr;
91          queue_node *tail = nullptr;
92      };
```

- Lines 38-40: We declare layout of the `queue_node` structure. It stores an integer `value` and a pointer to the `next` node in the list.
- Lines 44-57: We implement `push()` method which allocates new node and sets its value.
- Lines 59-75: We implement `pop()` method which deletes the first element in the queue.
- Lines 77-87: The `show()` method walks the list and prints the contents of each node to standard out.

The preceding queue implementation stores values of type `int` in a linked list and provides three basic methods: `push()`, `pop()`, and `show()`.

In this section, we will demonstrate how to modify your volatile structure to store elements in persistent memory with `libpmemobj-cpp` bindings. All the modifier methods should provide atomicity and consistency properties which will be guaranteed by the use of transactions.

Changing a volatile application to start taking advantage of persistent memory should rely on modifying structures and classes with only slight modifications to functions. We will begin by modifying the queue_node structure by changing its layout as shown in Listing 8-7.

Listing 8-7. A persistent queue implementation – modifying the queue_node struct

```
38    #include <libpmemobj++/make_persistent.hpp>
39    #include <libpmemobj++/p.hpp>
40    #include <libpmemobj++/persistent_ptr.hpp>
41    #include <libpmemobj++/pool.hpp>
42    #include <libpmemobj++/transaction.hpp>
43
44    struct queue_node {
45        pmem::obj::p<int> value;
46        pmem::obj::persistent_ptr<queue_node> next;
47    };
48
49    struct queue {
...
100   private:
101       pmem::obj::persistent_ptr<queue_node> head = nullptr;
102       pmem::obj::persistent_ptr<queue_node> tail = nullptr;
103   };
```

As you can see, all the modifications are limited to replace the volatile pointers with `pmem:obj::persistent_ptr` and to start using the `p<>` property.

Next, we modify a `push()` method, shown in Listing 8-8.

Listing 8-8. A persistent queue implementation – a persistent push() method

```
50        void
51        push(pmem::obj::pool_base &pop, int value)
52        {
53            pmem::obj::transaction::run(pop, [&]{
54                auto node = pmem::obj::make_persistent<queue_node>();
55                node->value = value;
```

```
56                 node->next = nullptr;
57
58                 if (head == nullptr) {
59                     head = tail = node;
60                 } else {
61                     tail->next = node;
62                     tail = node;
63                 }
64             });
65         }
```

All the modifiers methods must be aware on which persistent memory pool they should operate on. For a single memory pool, this is trivial, but if the application memory maps files from different file systems, we need to keep track of which pool has what data. We introduce an additional argument of type `pmem::obj::pool_base` to solve this problem. Inside the method definition, we are wrapping the code with a transaction by using a C++ lambda expression, `[&]`, to guarantee atomicity and consistency of modifications. Instead of allocating a new node on the stack, we call `pmem::obj::make_persistent<>()` to transactionally allocate it on persistent memory.

Listing 8-9 shows the modification of the `pop()` method.

Listing 8-9. A persistent queue implementation – a persistent pop() method

```
67         int
68         pop(pmem::obj::pool_base &pop)
69         {
70             int value;
71             pmem::obj::transaction::run(pop, [&]{
72                 if (head == nullptr)
73                     throw std::out_of_range("no elements");
74
75                 auto head_ptr = head;
76                 value = head->value;
77
78                 head = head->next;
79                 pmem::obj::delete_persistent<queue_node>(head_ptr);
80
```

```
81                if (head == nullptr)
82                    tail = nullptr;
83            });
84
85            return value;
86        }
```

The logic of `pop()` is wrapped within a `libpmemobj-cpp` transaction. The only additional modification is to exchange call to volatile delete with transactional `pmem::obj::delete_persistent<>()`.

The `show()` method does not modify anything on either volatile DRAM or persistent memory, so we do not need to make any changes to it since the `pmem:obj::persistent_ptr` implementation provides `operator->`.

To start using the persistent version of this queue example, our application can associate it with a `root` object. Listing 8-10 presents an example application that uses our persistent queue.

Listing 8-10. Example of application that uses a persistent queue

```
39  #include "persistent_queue.hpp"
40
41  enum queue_op {
42      PUSH,
43      POP,
44      SHOW,
45      EXIT,
46      MAX_OPS,
47  };
48
49  const char *ops_str[MAX_OPS] = {"push", "pop", "show", "exit"};
50
51  queue_op
52  parse_queue_ops(const std::string &ops)
53  {
54      for (int i = 0; i < MAX_OPS; i++) {
55          if (ops == ops_str[i]) {
56              return (queue_op)i;
```

```
57              }
58          }
59          return MAX_OPS;
60      }
61
62      int
63      main(int argc, char *argv[])
64      {
65          if (argc < 2) {
66              std::cerr << "Usage: " << argv[0] << " path_to_pool"
                << std::endl;
67              return 1;
68          }
69
70          auto path = argv[1];
71          pmem::obj::pool<queue> pool;
72
73          try {
74              pool = pmem::obj::pool<queue>::open(path, "queue");
75          } catch(pmem::pool_error &e) {
76              std::cerr << e.what() << std::endl;
77              std::cerr << "To create pool run: pmempool create obj
                --layout=queue -s 100M path_to_pool" << std::endl;
78          }
79
80          auto q = pool.root();
81
82          while (1) {
83              std::cout << "[push value|pop|show|exit]" << std::endl;
84
85              std::string command;
86              std::cin >> command;
87
88              // parse string
89              auto ops = parse_queue_ops(std::string(command));
90
```

```
91            switch (ops) {
92                case PUSH: {
93                    int value;
94                    std::cin >> value;
95
96                    q->push(pool, value);
97
98                    break;
99                }
100               case POP: {
101                   std::cout << q->pop(pool) << std::endl;
102                   break;
103               }
104               case SHOW: {
105                   q->show();
106                   break;
107               }
108               case EXIT: {
109                   exit(0);
110               }
111               default: {
112                   std::cerr << "unknown ops" << std::endl;
113                   exit(0);
114               }
115           }
116       }
117   }
```

The Ecosystem

The overall goal for the libpmemobj C++ bindings was to create a friendly and less error-prone API for persistent memory programming. Even with persistent memory pool allocators, a convenient interface for creating and managing transactions, auto-snapshotting class templates and smart persistent pointers, and designing

an application with persistent memory usage may still prove challenging without a lot of niceties that the C++ programmers are used to. The natural step forward to make persistent programming easier was to provide programmers with efficient and useful containers.

Persistent Containers

The C++ standard library containers collection is something that persistent memory programmers may want to use. Containers manage the lifetime of held objects through allocation/creation and deallocation/destruction with the use of allocators. Implementing custom persistent allocator for C++ STL (Standard Template Library) containers has two main downsides:

- Implementation details:
  - STL containers do not use algorithms optimal for a persistent memory programming point of view.
  - Persistent memory containers should have durability and consistency properties, while not every STL method guarantees strong exception safety.
  - Persistent memory containers should be designed with an awareness of fragmentation limitations.
- Memory layout:
  - The STL does not guarantee that the container layout will remain unchanged in new library versions.

Due to these obstacles, the `libpmemobj-cpp` contains the set of custom, implemented-from-scratch, containers with optimized on-media layouts and algorithms to fully exploit the potential and features of persistent memory. These methods guarantee atomicity, consistency, and durability. Besides specific internal implementation details, `libpmemobj-cpp` persistent memory containers have a well-known STL-like interface, and they work with STL algorithms.

Examples of Persistent Containers

Since the main goal for the `libpmemobj-cpp` design is to focus modifications to volatile programs on data structures and not on the code, the use of `libpmemobj-cpp` persistent containers is almost the same as for their STL counterparts. Listing 8-11 shows a persistent vector example to showcase this.

Listing 8-11. Allocating a vector transactionally using persistent containers

```
33  #include <libpmemobj++/make_persistent.hpp>
34  #include <libpmemobj++/transaction.hpp>
35  #include <libpmemobj++/persistent_ptr.hpp>
36  #include <libpmemobj++/pool.hpp>
37  #include "libpmemobj++/vector.hpp"
38
39  using vector_type = pmem::obj::experimental::vector<int>;
40
41  struct root {
42          pmem::obj::persistent_ptr<vector_type> vec_p;
43  };
44
        ...
63
64      /* creating pmem::obj::vector in transaction */
65      pmem::obj::transaction::run(pool, [&] {
66          root->vec_p = pmem::obj::make_persistent<vector_type>
            (/* optional constructor arguments */);
67      });
68
69      vector_type &pvector = *(root->vec_p);
```

Listing 8-11 shows that a `pmem::obj::vector` must be created and allocated in persistent memory using transaction to avoid an exception being thrown. The `vector` type constructor may construct an object by internally opening another transaction. In this case, an inner transaction will be flattened to an outer one. The interface and semantics of `pmem::obj::vector` are similar to that of `std::vector`, as Listing 8-12 demonstrates.

Listing 8-12. Using persistent containers

```
71          pvector.reserve(10);
72          assert(pvector.size() == 0);
73          assert(pvector.capacity() == 10);
74
75          pvector = {0, 1, 2, 3, 4};
76          assert(pvector.size() == 5);
77          assert(pvector.capacity() == 10);
78
79          pvector.shrink_to_fit();
80          assert(pvector.size() == 5);
81          assert(pvector.capacity() == 5);
82
83          for (unsigned i = 0; i < pvector.size(); ++i)
84              assert(pvector.const_at(i) == static_cast<int>(i));
85
86          pvector.push_back(5);
87          assert(pvector.const_at(5) == 5);
88          assert(pvector.size() == 6);
89
90          pvector.emplace(pvector.cbegin(), pvector.back());
91          assert(pvector.const_at(0) == 5);
92          for (unsigned i = 1; i < pvector.size(); ++i)
93              assert(pvector.const_at(i) == static_cast<int>(i - 1));
```

Every method that modifies persistent memory containers does so inside an implicit transaction to guarantee full exception safety. If any of these methods are called inside the scope of another transaction, the operation is performed in the context of that transaction; otherwise, it is atomic in its own scope.

Iterating over `pmem::obj::vector` works exactly the same as `std::vector`. We can use the range-based indexing operator `for` loops or iterators. The `pmem::obj::vector` can also be processed using `std::algorithms`, as shown in Listing 8-13.

Listing 8-13. Iterating over persistent container and compatibility with STD algorithms

```
95        std::vector<int> stdvector = {5, 4, 3, 2, 1};
96        pvector = stdvector;
97
98        try {
99            pmem::obj::transaction::run(pool, [&] {
100               for (auto &e : pvector)
101                   e++;
102               /* 6, 5, 4, 3, 2 */
103
104               for (auto it = pvector.begin();
                  it != pvector.end(); it++)
105                   *it += 2;
106               /* 8, 7, 6, 5, 4 */
107
108               for (unsigned i = 0; i < pvector.size(); i++)
109                   pvector[i]--;
110               /* 7, 6, 5, 4, 3 */
111
112               std::sort(pvector.begin(), pvector.end());
113               for (unsigned i = 0; i < pvector.size(); ++i)
114                   assert(pvector.const_at(i) == static_cast<int>
                      (i + 3));
115
116               pmem::obj::transaction::abort(0);
117           });
118       } catch (pmem::manual_tx_abort &) {
119           /* expected transaction abort */
120       } catch (std::exception &e) {
121           std::cerr << e.what() << std::endl;
122       }
123
```

```
124         assert(pvector == stdvector); /* pvector element's value was
            rolled back */
125
126         try {
127             pmem::obj::delete_persistent<vector_type>(&pvector);
128         } catch (std::exception &e) {
129         }
```

If an active transaction exists, elements accessed using any of the preceding methods are snapshotted. When iterators are returned by `begin()` and `end()`, snapshotting happens during the iterator dereferencing phase. Note that snapshotting is done only for mutable elements. In the case of constant iterators or constant versions of indexing operator, nothing is added to the transaction. That is why it is essential to use `const` qualified function overloads such as `cbegin()` or `cend()` whenever possible. If an object snapshot occurs in the current transaction, a second snapshot of the same memory address will not be performed and thus will not have performance overhead. This will reduce the number of snapshots and can significantly reduce the performance impact of transactions. Note also that `pmem::obj::vector` does define convenient constructors and compare operators that take `std::vector` as an argument.

Summary

This chapter describes the `libpmemobj-cpp` library. It makes creating applications less error prone, and its similarity to standard C++ API makes it easier to modify existing volatile programs to use persistent memory. We also list the limitations of this library and the problems you must consider during development.