1 Background

Initially there were two proposals.

- Move the open counter from the device class to the top level io class
- Use shared_ptr

Initially both of these methods seems reasonable.

Shared pointers accomplishes the same goal with a control block changing pointer-to-implementation in to shared_ptr-to-implementation instead. One issue with shared pointers, however, was the inclusion of type information and a file descriptor id. A specialized shared pointer (i.e. not std::shared_ptr!) could allow modest metadata to be stored with the control block in addition to the pointer and reference count. A deeper review of the type hierarchy, however, revealed intrusive lists, implementable subtypes, and perhaps other forms of data. This means a including a shared_ptr to the impl in the IO object is not feasible.

Open Counter: This is essentially the same concept as above, but when the file is opened (or dup/dup2 is called) the counter is incremented. Calls to close are deferred until the counter returns to 0. Simple. Straightforward. Uncomplicated. However there is a thread safety issue insofar as the pointer while in current use. (Consider the posix case where it obtains a pointer, io*, which can become invalid while in active use. Even lockable variants also risk the handle becoming invalid while actively locked.)

	shared_ptr to implementation	**open|dup → close counting**
Multiple references to same io object	yes	yes
Dangling pointers	no	yes
Feasible?	requires significant changes	yes, very
Explicit Close Safe	possibly 2-stage release → closed, released (to pool, if desired)	yes - required
Implicit Close (C++)	Yes	No
Posix C layer		
is internal io* pointer thread safe?	yes	possibly (requires manual reference counting)
Ambiguity between c and C++ close	No	Yes i.e. io* opened in C++ application code, but closed in C

What to do? *The Intrusive Reference!*

Let the io class manage the reference counts while ioref works much like shared_ptr except it calls refInc/refDec on a pointer to the io class.

```
os::posix::ioref<os::posix::tty> mytty;
mytty = os::posix::open ("/dev/tty0", 0);
```

Since C++ usage does not require an assigned file descriptor, the C++ class need not maintain an associated file descriptor. The descriptor manager is modified to assign io* to an ioref<> array. In order to create/obtain a file descriptor usable in C, a new API is added to obtain a new file descriptor (much like the "dup" API provides). This could be implemented in the io object or separately:

int io::dup(int fd=-1);

static int io::dup(ioref<>& file, int fd=-1);

When called, it will assign an IO object to the requested descriptor slot(if >=0) or the nearest unused descriptor >2 (if available). This also requires that when a file descriptor is created, the developer would be required to explicitly call C-posix's close API.

Fd	Initial Value (ioref<>)		Operation	
1	io{count = 2, ...}	← new_io{...}	new_io->dup(1)	File descriptor 1 is reassigned to stdout. The previously held io object counter is decremented to 1 living reference.
2	io{count = 1, ...}	← new_io{...}	new_io->dup(2)	File descriptor 2 is reassigned to stderr. Since the previously reference io object no longer have references it should be closed and any resources are released back in to the free pool (or deleted).
3	io{count = 1, ...}			
4		← new_io{...}	new_io->dup()	Without a specified descriptor index, assign to the first unclaimed ioref.
				...etc...
10	io{count = 1, ...}	← nullptr or ioref::reset	close(10)	An existing file descriptor is closed.

I completed a very tentative test using an basic ioref<> pointer and it worked. I did not integrate with the posix-c code, though. Fully implementing this is a bit of an effort.

2 IO object state needs elaboration

Currently, IO objects appear to be binary – either in use or not in use. This is controlled exclusively by open and close. A new state is needed "inactive but not freed".

The design should be modified to preserve the IO object until all references to it are released while still permitting the resource to be closed. The one exception we can make is that, by the intrusive design, a previously closed IO object can become valid if it is opened again. This can only be resolved if IO instances aren't linked to specific implementations. (In other words, no more implementable/lockable variants or intrusive containers.)

Note: I did see some code commented out referring to a do_release so I think there was some anticipation that it would become needed.

3 Special Cases

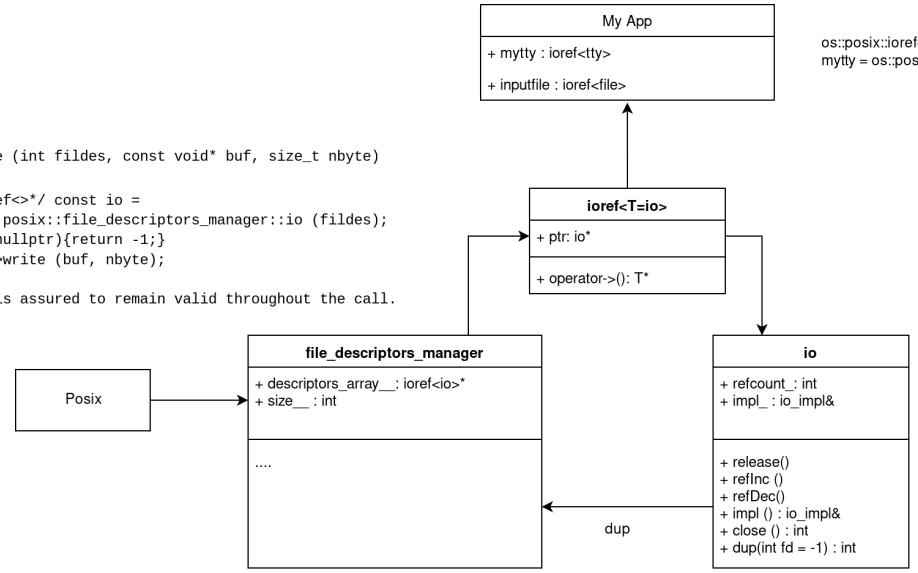
There are some special cases - a open counter may still be required in many cases. The intrusive pointer could be made aware of it's special status to ensure open/dup'd references are handled properly.

A second special case may involve the need to maintain a pointer to itself using a weakly linked concept. This doesn't translate well to a intrusive references. (It works well for shared pointers because of the minimal memory consumed by a control block.)

4 Decoupling Posix C and Posix C++

The proposed change to the design can permit the posix C backend can be optionally included. In addition, by removing the file descriptor from the C++ objects, C++ code is no longer constrained to the size of the descriptor table.

5 Here is a diagram!




```
ssize_t
__posix_write (int fildes, const void* buf, size_t nbyte)
{
  auto /*ioref<>*/ const io =
            posix::file_descriptors_manager::io (fildes);
  if (io == nullptr){return -1;}
  return io->write (buf, nbyte);
}
The pointer is assured to remain valid throughout the call.
```

```
ioref<tty> mytty = ...;
ioref<char> chardev = ...;
chardev->dup(1)
chardev = nullptr OR chardev.reset()
   chardev is still alive as a file descriptor
file->dup() -> allocates file descriptor -> 3 (i.e. reserved__)
file->dup(1) -> assigns to stdout,
   chardev is swapped with file, and released (implies close)

close(3);
close(1); --->
    file_descriptors_manager::deallocate(fd) {
        descriptors_array.reset()
        // may cause closed to be called
    }
```