The routine `psMemCheckLeaks` may be used to check for memory leaks. The return value is the number of blocks that have been allocated but not freed. Only blocks with `psMemBlock.id` greater than `id0` are checked; this allows the user to ignore blocks allocated by initialization routines.

If the argument `array` is non-`NULL`, then `**array` is set to an array of `psMemBlock *` pointers when the function returns. These pointers represent the blocks which have been allocated but not freed. It is the caller's responsibility to free this array with `psFree`. Also note that `**array` should be `NULL` (or not point to allocated memory) upon entering the call or the corresponding memory reference will be lost.

If the argument `fd` is non-`NULL`, a one-line summary of each block that has been allocated but not freed is written to that file descriptor.

The routine `psMemCheckCorruption` checks the entire heap for corruption, calling the routine registered with `psMemProblemCBSet` for each block detected as being corrupted. The return value is the number of corrupted blocks detected. If the argument `abort_on_error` is true, `psMemCheckCorruption` shall call `psAbort` as soon as memory corruption is detected.

2.1.7 Reference Counting

As mentioned above, the memory management system provides the `refCounter` element in `psMemBlock` to allow for the management of multiple references to the same block of memory. External APIs which make use of this structure must increment the counter for every additional reference to an allocated memory block, and decrement it when those references are removed. The memory management routines respect the use of the `refCounter` field: `psFree` will not free a block for which `refCounter != 1`, and `psAlloc` will initialize the field to 1. `psFree` must generate an error if `refCounter != 1`. However, they do not (and cannot practically) enforce the use of the counters; this is a requirement of external APIs which intend to use the feature.

Several APIs are provided to manage the reference counters. These APIs are:

```
int psMemGetRefCounter(void *vptr);
void *psMemIncrRefCounter(void *vptr);
void *psMemDecrRefCounter(void *vptr);
```

The functions all take a pointer to the start of a user block of memory. The first simply returns the value of the reference counter. The next two functions increment or decrement the reference counter, returning the pointer which was passed in. These functions must validate the memory pointer by determining the corresponding `psMemBlock.id` and checking for consistency in the internal memory block table (the table pointer for `psMemBlock.id` should be in the valid range and should correspond to the address of the `psMemBlock`). For an example implementation of the `refCounter` facilities, see the discussion of `psDlist` (§3.4).

2.1.8 Relation of Memory Management to Structures

In this document, we specify several C `structs`. It is expected that instances of, for example, `struct psMyType` will be constructed using `psMyTypeAlloc()` calls, and destroyed using `psMyTypeFree()` calls. The allocator will allocate the required memory with `psAlloc` and increment the appropriate `refCounter`.

The existence of complicated structures which include pointers to other structures require that we lay out a rule regarding destructors (i.e., `psMyTypeFree`) and reference counters. Simply put, *the destructor for every structure should only call* `psFree` *if* `refCounter == 1`*; otherwise, it decrements the reference counter and returns.* An example destructor is shown below:

```
void psMyTypeFree(psMyType *myType ///< Object to destroy
    )
{
    /* No operation if object is NULL */
    if (myType == NULL) {
return;
    }
    /* Only call psFree if reference counter is 1 */
    if (psMemGetRefCounter(myType) == 1) {
psFree(myType);
return;
    }
    /* Otherwise, decrement the reference counter only */
    psMemDecrRefCounter(myType);
}
```

This allows, for example, the `psMyType` to be imported into the metadata (§5.2) without the user worrying about the details of the memory allocation/deallocation:

```
void psFooMetadata(psMetadata *md)
{
    psFoo *foo = psFooAlloc();
    (void) psMetaDataAppend(md, psMetaDataItemAlloc(PS_META_FOO,foo,"Comment","foo.bar"));
    (void) psFooFree(foo);
}
```

In the above case, `foo` is created, stuffed into the metadata, and then the programmer follows the rule of "for every `alloc`, there is an equal and opposite free" before the function returns. However, the metadata needs to carry around the `psFoo`, and so it is important that `psFooFree` does not free the memory for `foo`, but only decrements its `refCounter`. Hence, at the conclusion of the function, the memory pointed to by `foo` in the course of the function remains allocated, and the corresponding `refCounter` is 1 (specifically, the reference in the metadata).

2.2 Tracing and Logging

This section defines the Pan-STARRS Tracing and Logging APIs; the former refers to debug information that we wish to be able to turn on and off without recompiling (it will *not* be available in production code); the latter means information about the processing that must be collected and saved, even in the production system. We envision that extensive use will be made of `psTrace` throughout the Pan-STARRS code.

2.2.1 Tracing APIs

A code-tracing facility should allow the programmer to place messages in the code which, when called, will print some useful information about the containing block of code. Ideally, different messages may be specified to have different levels (of severity or interest). For a given run of the program, the level of interest should be set to provide more or less feedback, depending on the needs of the programmer. In a typical situation, low-level messages would be placed generously throughout the code, indicating the flow of the program. Higher-level