TODO

* Global field name – `gc` or `GC`?
* Determine the exact needs of the new set of TypeInfo classes
* Allocator free – inner blocks? <- No idea what I meant here, ~~perhaps considering a way to determine if a pointer was an internal block or not~~ already included in a later TODO?
* init/shutdown API – need to be global
* Aligned allocation
* Optional generation of scanning code rather than bitmaps? If so, **MUST** be shared globally, perhaps represent bitmap as a base64-esqu symbol name?
* stopIgnoringThread?
* Look at destructor handling again, original design failed to account for user attempts to invoke the destructor of a non-GC allocated object.
* Change the way allocators are setup – Each allocator no longer owns just one page, may own many pages.
* Reserve & (extend &| realloc) API
* Add ways to obtain information about allocations:
  + Is an inner slice?
  + Total size?
  + Base pointer?
  + Extendable amount?

# Goals

* Speed – Above all, the GC needs to be fast, if this means that some code will need to be dynamically generated to avoid branch mispredictions, so be it.
* Minimize Impact – The GC needs to strive to achieve the minimum possible impact on the final end-user code by using strategies such as concurrent marking and preemptive collections.
* Extensible – The GC needs to be able to coexist both peacefully, and cleanly, with Andrei’s allocators.

# Constraints

* While it is allowed to change the internal API of the GC drastically, and silently, it must be almost taboo to make a silent breaking change for user code. It is however perfectly allowed to make a nice loud breaking change for user code.
* While grammar extensions are welcome, grammar changes are heavily discouraged.

# Term Notes

* In the scope of this document, **finalizers** and **destructors** are used interchangeably to refer to the same thing. No distinction is made in terminology between heap and stack allocated value finalization.
* Unless otherwise obvious, the term **allocator** refers to GC allocators.

# Major Differences

* There is no way to allocate initialized memory; instead the caller deals with initializing the memory.
* Due to the need to be able to work with a unified type info interface for structs, delegates, arrays, classes, pointers, etc., the current `TypeInfo` classes will, for the most part, be thrown out the window, in favor of a much more unified design.
* All objects are guaranteed to have their finalizers called before the program exits, provided that the program is not terminated abnormally. The last few (up to the number of cores - 1) threads left running may be hijacked to run any remaining finalizers before they are allowed to exit along with the main thread.
* Allocations and exceptions in finalizers are allowed, however please note that exceptions will trigger a fatal error unless a user-defined handler is set.
* Finalizers for heap-allocated structs will be invoked.
* Destructors will have 2 possible places in the method where it can start, the first makes a call to `gc.markFinalized`, and, if it returns true, will continue with the rest of the body, and otherwise return, the second, the one that any stack allocated value will invoke, will skip the call to `gc.markFinalized`, and instead go directly to the body of the finalizer.

# Concessions Made

* Compacting will not be supported with the current design. It is feasible to modify the API to work with a compacting GC, but the time investment required for the design is currently better spent designing the core of the GC that will actually be used. The API does however implement what is required to pin an object from user code, that is, to ensure it is not moved. Please note that while the pinning API will currently do absolutely nothing, that may be changed in the future, and user code should be written under the assumption that the pinning API does indeed function, and is required.

# Allocator Dispatch

The outermost layer of the GC is the Allocator Dispatch. It manages the interaction with the OS, freeing empty pages, when to collect, and if a precise or conservative collection should be performed. It is also the only part of the GC that external D code will be interacting with. The default page size is 4mb, but this may be scaled up or down at compile-time depending on the target memory environment. The Dispatch itself is to be completely non-locking, with lock-prefixed instructions where required. Dirty reads are to be preferred over a write-locked read. This is not required of allocator implementations, it is however preferred. The Dispatch will implement a modest-sized API that is detailed, in no particular order, as follows.

1. A public, null-initialized field of the type `GC` by the name of `gc`
2. A public final *class* by the name of `GC` which has the following instance members:
   1. Method - addRoot
   2. Method - addRootRange
   3. Method - removeRoot
   4. Method - removeRootRange
   5. Method - minimize - return all free pages to OS
   6. A method by the name of `pinAllocation` accepting a single, potentially interior, pointer to an allocation. It is to be an error to call this with a pointer that is either not allocated by the GC, or else was already freed by the GC. For each invocation of this method referencing an allocation, a call to `unpinAllocation` must be performed in order to allow the underlying allocator to once again move the allocation. This method will ensure that if a compacting allocator was used to create the allocation, the allocation will not be moved. Please note that this **will not** prevent an allocation from being collected. It is an error for a pinned allocation which is not a root to have no living references. Please note that this will not prevent it from being copied by a call to realloc/extend, as it is assumed that the caller of those methods will update whatever references to the allocation are required.
   7. A method by the name of `unpinAllocation` accepting a single, potentially interior, pointer to an allocation. It is to be an error to call this with a pointer that is either not allocated by the GC, or else was already freed by the GC. Provided that this method has been called exactly once for every call to `pinAllocation`, it will tell a compacting allocator that it is once again allowed to move the allocation.
   8. A method by the name of `ignoreThread` accepting a single `tid\_t` which will tell the GC to neither stop, nor scan the thread. An example of a thread that would be suitable to being marked as not scanned would be an external thread that does only manual memory management, and will not have any pointers to objects allocated by D that have not been added as roots. *This will be one of the instances of dynamic code-gen with a fallback for unknown platforms. Due to the fact that there should only be a small number of threads being ignored by the GC, we will statically expand all comparisons into pairs of cmp & setne instructions, this will eliminate all possible cases where the branch or even the return could be mispredicted by the CPU.*
   9. Method - unIngoreThread
   10. A method by the name of `markAsReferenced` accepting a single pointer to a potential GC allocation. This method will determine which allocator, if any, owns the pointer, and will invoke that allocator’s `markAsReferenced` method, passing the pointer as-is.
   11. A method by the name of `finalize` accepting a pointer to a heap-allocated value, and a pointer to the type info for that value. This will return true if the value has no finalizer to call, meaning that the allocator that called this is able to free the memory used by this allocation immediately. If this method returns false, the allocator that calls this must not mark the memory used by the allocation as free, but must ensure that it is not freed, and must also ensure that it is only ever passed to this method once, otherwise multi-finalization will occur.
   12. A method by the name of `markAsFinalized` accepting a single pointer to the allocation to mark as finalized and returning true if the value was successfully marked as finalized, and false in any other case, such as when it has already been finalized by user-code before. This will determine which allocator owns the allocation and call that allocator’s `markAsFinalized` method. If this method returns false, then the body of the finalizer should not be invoked. This shall **not** be called by the GC itself, nor by user code; instead it shall be called only by the prologue generated for finalizers.
   13. A method by the name of `releaseAllocator` accepting a single pointer to the allocator to release. The allocator being released is responsible for ensuring that all allocations it owns have been properly finalized and are indeed free before calling this. It is not guaranteed that an allocator passed into this will be immediately released, because it may be cached for future use depending on the volume of available memory.
   14. Method - allocate
   15. Method - allocateArray
   16. Method - free
   17. A method by the name of `collect` which will trigger a full collection, and will block until both the mark and sweep passes have finished. If a truly complete collection is desired, and there are a large number of objects requiring finalization, such as may be the case when changing maps in a game, it is suggested to call this method, then `waitForPendingFinalizers`, then finally this method again so that the space used by those objects with finalizers is actually released. It is not recommended for non-finalizer heavy code to do this, as it will trigger 2 full collections.
   18. A method by the name of `waitForPendingFinalizers` which will block this thread until all pending finalizers have been invoked. It is likely that this method will use the invoking thread to invoke some of the pending finalizers, so it should not be assumed that a thread calling this method will go to sleep.
   19. A method by the name of `takeOwnership` accepting a pointer to a vtable containing pointers to, in this order, `markAsReferenced`, `sweep`, `free`, and `markAsFinalized`, implemented as described for GC allocators. Followed by a pointer to the location in memory to mark as owned, and a `size\_t` representing the length of the block of memory to take ownership of. This method will tell the GC that pointers to values in the provided memory range should be handled by invoking methods in the provided vtable pointer. A `MemoryAlreadyOwned` error will be thrown if the memory location is already owned by another allocator. This error may be handled by user code with the `onMemoryAlreadyOwnedError` delegate.
   20. A method by the name of `releaseOwnership` accepting a pointer to a memory location that has previously been passed to `takeOwnership` This method will tell the GC that it should no longer assume that the previously provided allocator owns pointers to values in the memory range. Allocator implementations should not return the memory to the OS until this method has returned, to avoid the possibility of it being re-used by another allocator before the ownership of the memory has been removed.
   21. A method by the name of `enable` that will enable automatic garbage collection if it has previously been disabled by a call to `disable`. It is required to call this method once for every call to `disable`. A `GCAlreadyEnabled` error shall be thrown if this method is called when the GC is already enabled, and can be handled through the `onGCAlreadyEnabledError` delegate. This is to prevent code which has imbalanced enable/disable logic from silently succeeding when the GC is already enabled.
   22. A method by the name of `disable` that will disable automatic garbage collection. It is required to call `enable` once for every call to this method in order to enable the GC again. This is done to allow specific functions in large libraries (**not Phobos**) to disable and re-enable the GC for whatever reason they wish, without causing any code invoking it, which may also have a reason to disable the GC, to have to re-disable the GC after calling that method.
   23. A `bool` getter property method by the name of `enabled` that will return `true` if the GC is currently enabled and may trigger a collection, otherwise it will return false.
   24. A field by the name of `onDisposalError` containing a delegate accepting a single parameter of the type `DisposalError` that will be invoked when an exception occurs while calling the finalizer of an allocated value. The default handler simply throws the error passed to it.
   25. A field by the name of `onFreeOwnershipError` containing a delegate accepting a single parameter of the type `FreeOwnershipError` that will be invoked if an allocator’s `free` is passed a pointer to an allocation for memory not actually owned by the allocator. This should never happen in a properly functioning environment, but, as the allocators will throw the error, there needs to be a way to catch it.
   26. Field - onMemoryAlreadyOwnedError
   27. Field - onGCAlreadyEnabledError
   28. Field - onCollect - Called **after** collection & sweep with stats
   29. A field by the name of `asyncSweepEnabled` containing a `bool` which determines if the GC will attempt to perform an asynchronous sweep, allowing all non-blocked threads to continue while the sweep is occurring, as well as the lone potentially blocked thread when it’s able to make the allocation successfully.
   30. A field by the name of `preemptiveCollectionsEnabled` containing a `bool` which determines if the GC is allowed to preemptively schedule a collection, to avoid blocking a future allocation request due to the need to run a collection pass.
   31. Field - ***stats***?

# The Allocators

The first distinction that needs to be made is that these are **not** Andrei’s allocators. It is intended for Andrei’s allocators to operate on top of these allocators.

Every page gets its own allocator, which is allocated, with an expanded vtable, at the start of the page. An allocator handles how the actual objects are allocated and freed within it. Allocators can differ in multiple ways, including, but not limited to, the allocation strategy, the freeing strategy, how it handles references and the scanning methods used. Each allocator provides a basic interface, detailed as follows, in this order.

1. A volatile, atomic **signed** `size\_t` by the name of `largestAvailableAllocation` representing the largest available contiguous blog of memory currently possible to be allocated within the page. While the Allocator Dispatch will never attempt to allocate a value in a specific allocator if the size of the object is greater than this value, due to fact that it is a volatile value, it may have read the value before it was modified, meaning that an allocator cannot assume that it will never be asked to allocate a value larger than this. It is recommended for an allocator to only calculate this value after it has failed to perform a requested allocation. It is also recommended to immediately subtract the size of the object being allocated from this value immediately at the start of attempting an allocation, and, if failing to allocate it, re-add that size back, to reduce the instances of lock contention when multiple threads are allocating large numbers of medium sized objects. It is valid for this value to be less than zero. It is also required to recalculate this value during a call to `sweep`. It is preferred for this value to be calculated procedurally during a call to `sweep` to allow unblocking the thread that triggered the collection sooner. It is not required, and is discouraged, to recalculate it after a call to `free`.
2. A method by the name of `markAsReferenced` accepting a pointer to an object, or a pointer to the interior of an object that, if valid, lies within the allocator’s page. This should mark the object, and any heap-allocated value referenced by the object by passing a pointer to the object to `gc.markAsReferenced`, so that a subsequent call to `sweep` doesn’t result in one of the values referenced by this object being freed by another allocator. This should treat the final possible pointer in an allocation as a special case to allow the compiler to perform tail-call optimizations.
3. A method by the name of `allocate` accepting a pointer to the type info of the object to allocate, and returning either a pointer to the allocated value, or else `size\_t.max`. Any other value returned is considered a valid allocation. It is **required** for allocator implementations to retain this value, as it contains information essential to the correct functioning of the GC, including, but not limited to, the finalizer of an allocation.
4. A method by the name of `allocateArray` accepting a `size\_t` representing the size of the array to allocate in **bytes**, and a pointer to the type info of the array itself. Its return value and retention of the type info pointer should be consistent with `allocate`.
5. A method by the name of `sweep` accepting no parameters, returning a `size\_t` with the number of objects freed. This return value will only ever be used for statistics, so, while it is encouraged for an allocator to implement this, it is permissible to unconditionally return `0`, as the return value will never be essential to the functioning of the GC. This method is to be invoked only after every root has been recursively marked as referenced. An allocator is free to do whatever it wants with the freed allocations, however all freed allocations need to be passed to `gc.finalize` which will determine if the value is actually free due to pending finalizers. It is encouraged for allocator implementations to take lock contention into account when implementing this method, as the Allocator Dispatch may request an allocation while the sweep is still being performed. It is recommended for an allocator to treat a completely un-marked heap as a special case, so that, if there are finalizers pending, time is not wasted attempting to add them to the finalizer list. If there are no remaining live allocations belonging to an allocator, that allocator should pass a pointer to itself to `gc.releaseAllocator`, so that it can be released if it is no longer needed.
6. A method by the name of `free` accepting a single pointer to an allocation owned by this allocator. This method should assume that the finalizer has already been called if applicable, and is free to use the block of memory in whatever way it wants. It is not required for an allocator to immediately make the memory available to be re-allocated, and it is permitted for the allocator to wait until the next time that `sweep` is called to make the memory available for re-allocation.. While generally errors should not be thrown in the GC, if a pointer to an object is passed to an allocator’s free method that is not in fact owned by that allocator, the allocator should throw an Error, due to the fact there are likely even bigger problems. *It will be possible to catch this error via the global handler `onFreeOwnershipError`.*
7. A method by the name `markAsFinalized`
8. Method - pinAllocation
9. Method - unpinAllocation

# Random Performance Ideas

* Conditionally strongly hinted for loops – By using a separate non-hinted for loop for small iteration numbers, it would allow the CPU to eliminate the possibility of a misprediction, while using a strongly-hinted not-taken for loop condition branch for the array sizes that are too large to be accurately predicted by the CPU, which would make it so that there is exactly 1 branch misprediction for each array. *This will likely be a pointless optimization and it would be better to simply strongly hint the for loop of all array scanning, as the 1 branch misprediction per-array will very likely be out-weighed by the misprediction of the branch determining whether to take the unhinted path or not.*
* Prefetch T0 both sides of equally-likely, or known to both be taken, branches. This specifically includes adding a prefetch hint at the end of a for-loop body for the case where the condition fails.