Todo

**General**

* Rewrite this list. (After outlining a ‘definition of done’.)
* Todo list.
  + Consider porting to Google Code issue tracker.
* Re-architect to make mocking for tests easier.
* Split into sub-libraries where appropriate (winapi wrapper, memory editing, injected-only, stealth hooks, etc).
* Reorganize directory and build structure where appropriate.
* Document and check (with pp) minimum Boost and compiler verisons.
* Value-type style initialization rewrite.
* Document preconditions and postconditions.
* Better annotations (warnings, notes, etc) in docs.
* Check implementation of constructors in derived classes.
* Use Boost.Locale for i18n and l10n support.
* Revisit move/copy/etc constructors and ensure they are all implemented correctly.
  + Alternatively, only implement them explicitly when required, and document the supported ‘concepts’ in the library docs. E.g.CopyConstructible
* Compiler feature testing instead of version testing.
* Remove hard dependency on Boost and use environmental variables instead.
  + Do the same for other dependencies? Decouples dependencies in cases where projects dependent on HadesMem want to use the same dependencies without forcing them to use the version provided by HadesMem (HadesMem should instead use the version provided by the user, assuming it meets the minimum requirements).
* Improve build process.
* Investigate support for ICC, GCC, Clang, etc.
* Move ‘internal’ code to ‘detail’ folder/namespace.
* Documentation.
  + Improved and expanded documentation. Add notes, cautions, symbols, cross-refs, etc.
  + Threading docs.
  + Exception guarantee docs.
* Unit tests.
  + Extra sanity checks to ensure not only that functions return without exception, but also that the returned data is valid. (e.g. GetName in ImpThunk.)
  + Expand tests to be more thorough and use Boost.Test checks such as BOOST\_CHECK\_NO\_THROW, BOOST\_CHECK\_THROW, BOOST\_CHECK\_EQUAL\_COLLECTION, etc.
  + Extra tests on library usage that should fail to compile (because of static assertions, type traits, etc). Needed to ensure library misuse is not ‘possible’.
* Examples.
  + All components.
  + Cleanup and in-source documentation.
* Improve quality of existing code.
  + Exception safety ‘rewrite’.
  + Use ‘noexcept’ where appropriate.
  + Change equality comparison style in code to put constants on the left hand side.
  + Follow (mostly) Google C++ Style Guide, LLVM Style Guide, Chromium Style Guide, etc.
  + Virtual function ‘override’ annotations.
  + Replace enums with virtual functions.
  + Check default constructability etc in templates and add static asserts.
  + Improve exception-safety guarantees. (Rollback support etc)
  + Improve thread-safety guarantees and threading model.
  + Investigate Boost.Exception’s support for multiple exception info objects which share the same tag type, and how to ‘extract’ them at the catch site. Potentially rewrite some exceptions which use multiple ‘ErrorCodeWin’ objects to use unique tags instead.
  + Proper localization and internationalization support/framework.
  + Consider moving from UTF-16 internally to UTF-8, and only using UTF-16 at API boundaries. If implemented, all files and other IO should be in UTF-8.
  + Template concept checking and improved error messages.
  + Compile time checks to ensure MSVC, GCC, Boost, etc meet the minimum requirements.
  + Value-type style uniform initialization (e.g. std::set<T> = GetT()).
  + Assertions where appropriate (for ‘impossible’ or rare cases).
    - Especially check for things like integer overflows that would otherwise be hard to debug.
  + Support swapping where appropriate (and use to simplify operator= if possible).
  + Support extra operator overloads where appropriate (such as < or > on Module to make it closer to the built in HMODULE type).
  + Support perfect forwarding where appropriate.
  + Fix conversion to bool in classes (e.g. EnsureCleanup) to use an unspecified bool type and prevent implicit conversion to int, and also to remove the reliance on HandleType’s implicit conversion to bool, which we cannot guarantee.
  + Const correctness.
  + Clean up APIs returning pairs/tuples/etc to return objects with named fields and/or getters.
  + Review and ‘rewrite’ in-code documentation/comments.
  + Tests for remaining ‘detail’ components such as EnsureCleanup, StringBuffer, etc.
  + Investigate support for optional error reporting via error codes to avoid littering code with try-catch blocks in certain scenarios.
  + Improve API genericity.
    - Instead of taking or returning a vector<T>, instead use a C<T> where ‘C’ is a template parameter representing an arbitrary container.
    - Use ‘ranges’ over direct containers where possible.
* Reduce compile time.
  + Reduce include dependencies.
    - Pointer-to-impl idiom.
  + Decouple components if possible.
* Performance improvements in potential bottlenecks (e.g. Iterators, Scanner, PeLib, FindPattern, etc).

**New Modules**

* Basic base hook.
* Cheat Engine ‘replacement’.
* Helper service to run HadesMem tools as ‘SYSTEM’ for when manipulating certain protected/critical processes (running in separate desktops, sessions, etc.).
* Debugging.
  + Software breakpoint.
  + Hardware breakpoint (including memory read/write detection).
  + Conditional breakpoints.
  + Single stepping.
  + Stack trace.
* Anti-anti-cheat.
* Thread/process information and enumeration.
* Remote code ‘emulator’.
* Custom GetModuleHandle, GetProcAddress, etc.
* C++ WinAPI wrapper.

**MemoryMgr**

* Consider using array in Read over vector if possible (as sizeof(T) is a compile-time constant). Union template for { T obj; char bytes[sizeof(T); } also available.
* Use Boost.FunctionTypes in remote function caller to detect calling convention. (Also use TMP to detect number of args and their types etc).
* Improve ‘safety’ of remote function caller via EH to minimize risk of crashing the target.
* Improve genericity of parameter passing in remote function caller.
* Memory reading via expression templates.
* Support floating point parameters and return values in remote function caller.
* Support 64-bit parameters under x86 in remote function caller.
* Support non-MSVC compilers in remote function caller (e.g. in calling convention specification).
* Memory iterators/functors.

**ManualMap**

* Exception handling support under x86 SafeSEH and x64.
* Improved TLS support.
* CLR hosting support.

**Injector**

* .NET injection.
  + Without DLL dependency if possible.
* Cross-section injection.
* IAT injection.
* Get address of Kernel32!LoadLibrary ‘manually’ rather than using a local GetProcAddress and pointer arithmetic.
  + Whilst this works in all normal cases, it doesn’t work when the target has shims enabled which hook LoadLibrary.

**Patcher**

* Fix Patcher tests requiring UAC elevation. Embedding an ‘asInvoker’ manifest will stop the compat shims misfiring due to the filename.
* VEH hooking (both INT3 and DR).
* Transactional hooking.
* Improved relative instruction rebuilding (including conditionals). x64 has far more IP relative instructions than x86.
* Freeze target when hooking (except calling thread if applicable – e.g. in injected code).
* When hooking on x64 try to find a free memory block for the trampoline that is within RIP-relative range of the detour. Only if one cannot be acquired should we fall back to a system-provided address and an absolute jump.
* Uncopyable, so make moveable.
* VMT hooking.
* IAT/EAT hooking.
* Explicitly support hook chains (and write test).
* Use relative jumps where possible (detect delta at runtime).
* Detect cases where hooking may overflow past the end of a function, and fail. (Provide policy or flag to allow overriding this behaviour.) Examples may be instructions such as ‘int 3’, ‘ret’, ‘jmp’, etc.

**FindPattern**

* Pattern generator.
* ‘Multi-pass’ support (e.g. search for pattern, apply for manipulators, use as starting point for second search).
* Arbitrary region support.

**PeLib**

* Support for working on x86 PE files from x64 and vice versa.
* Investigate use of virtual functions for file vs memory access (RvaToVa).
  + Alternatively, investigate use of templates, which may ‘merge’ better with x86/x64 cross compatibility.
  + Note: May cause problems when copying ‘PeFile’ type.
* Extra sanity checking in all components.
  + E.g. Check NumberOfRvaAndSizes in NtHeaders before attempting to retrieve a data dir.
* Cache base pointers etc rather than retrieving it manually in every getter/setter. Slightly less ‘robust’, but due to the typically ‘read-only’ nature of the data this is the expected behaviour in all known cases anyway.
* Support more of the PE file format.
  + Overlay data.
  + Resource directory.
  + Exception directory.
  + Relocation directory.
  + Security directory.
  + Debug directory.
  + Load config directory.
  + Delay import directory.
  + Bound import directory.
  + IAT (as opposed to Import) directory.
  + CLR runtime directory support.
* Full support for writing back to PE file, including automatically performing adjustments where required to fit in new data or remove unnecessary space.
* Improve export forwarding code to detect and handle forward-by-ordinal explicitly rather than forcing the user to detect it and do string manipulation and conversion.
* Helper functions such as FindExport, FindImport, HasDataDir, GetArchitecture, IsDotNet, GetPDB, etc.
* Test against pathological cases such as Corkami tests.

**Disassembler**

* Decode calls/jumps to function names if possible.
* NOP/UnNOP support.
* ASM searching API with ‘wildcards’. (MetaASM?)
* String based assembler with x64 support.
* Disassemble function API.

**Scanner**

* Rewrite to be more reliable and robust.
* Refactor to reduce code duplication.
* Use a file view with a small memory cache rather than consuming large amounts of RAM.
* Multi-threaded scanning options.
* Wildcard support for vector/string scanning.
* Regex support for string scanning.
* Memory protection filters (read, write, exec).
* Memory type filters (private, mapped, image).
* Support pausing target while scanning.
* Support injected scanning.
* Configurable scan buffer size.
* Pointer scanner.
* Unknown value scan.
* Progressive scan filtering based on either value or criteria.
* Scan history and undo.
* Support case insensitive string scanning.
* Binary scanning.
* Custom scanning via user-supplied predicate.
* Improved floating point support (configurable or ‘smart’ epsilon).
* Group search support.