Last modified Mon Feb 14 2022

This is the heap checker we use at Google to detect memory leaks in C++ programs. There are three parts to using it: linking the library into an application, running the code, and analyzing the output.

Linking in the Library

The heap-checker is part of tcmalloc, so to install the heap checker into your executable, add `-ltcmalloc` to the link-time step for your executable. Also, while we don't necessarily recommend this form of usage, it's possible to add in the profiler at run-time using `LD_PRELOAD`:

```
% env LD_PRELOAD="/usr/lib/libtcmalloc.so"
```

This does *not* turn on heap checking; it just inserts the code. For that reason, it's practical to just always link `-ltcmalloc` into a binary while developing; that's what we do at Google. (However, since any user can turn on the profiler by setting an environment variable, it's not necessarily recommended to install heapchecker-linked binaries into a production, running system.) Note that if you wish to use the heap checker, you must also use the tcmalloc memory-allocation library. There is no way currently to use the heap checker separate from tcmalloc.

Running the Code

Note: For security reasons, heap profiling will not write to a file -- and is thus not usable -- for setuid programs.

Whole-program Heap Leak Checking

The recommended way to use the heap checker is in "whole program" mode. In this case, the heap-checker starts tracking memory allocations before the start of `main()`, and checks again at program-exit. If it finds any memory leaks -- that is, any memory not pointed to by objects that are still "live" at program-exit -- it aborts the program (via `exit(1)`) and prints a message describing how to track down the memory leak (using pprof).

The heap-checker records the stack trace for each allocation while it is active. This causes a significant increase in memory usage, in addition to slowing your program down.

Here's how to run a program with whole-program heap checking:

1. Define the environment variable HEAPCHECK to the type of heap-checking to do. For instance, to heap-check `/usr/local/bin/my_binary_compiled_with_tcmalloc`:

```
% env HEAPCHECK=normal /usr/local/bin/my_binary_compiled_with_tcmalloc
```

No other action is required.

Note that since the heap-checker uses the heap-profiling framework internally, it is not possible to run both the heap-checker and heap profiler at the same time.

Flavors of Heap Checking

These are the legal values when running a whole-program heap check:

1. `minimal`
2. `normal`
3. `strict`
4. `draconian`

"Minimal" heap-checking starts as late as possible in a initialization, meaning you can leak some memory in your initialization routines (that run before `main()`, say), and not trigger a leak message. If you frequently (and purposefully) leak data in one-time global initializers, "minimal" mode is useful for you. Otherwise, you should avoid it for stricter modes.

"Normal" heap-checking tracks live objects and reports a leak for any data that is not reachable via a live object when the program exits.

"Strict" heap-checking is much like "normal" but has a few extra checks that memory isn't lost in global destructors. In particular, if you have a global variable that allocates memory during program execution, and then "forgets" about the memory in the global destructor (say, by setting the pointer to it to NULL) without freeing it, that will prompt a leak message in "strict" mode, though not in "normal" mode.

"Draconian" heap-checking is appropriate for those who like to be very precise about their memory management, and want the heap-checker to help them enforce it. In "draconian" mode, the heap-checker does not do "live object" checking at all, so it reports a leak unless *all* allocated memory is freed before program exit. (However, you can use IgnoreObject() to re-enable liveness-checking on an object-by-object basis.)

"Normal" mode, as the name implies, is the one used most often at Google. It's appropriate for everyday heap-checking use.

In addition, there are two other possible modes:

- `as-is`
- `local`

`as-is` is the most flexible mode; it allows you to specify the various knobs of the heap checker explicitly. `local` activates the explicit heap-check instrumentation, but does not turn on any whole-program leak checking.

Tweaking whole-program checking

In some cases you want to check the whole program for memory leaks, but waiting for after `main()` exits to do the first whole-program leak check is waiting too long: e.g. in a long-running server one might wish to simply periodically check for leaks while the server is running. In this case, you can call the static method `HeapLeakChecker::NoGlobalLeaks()`, to verify no global leaks have happened as of that point in the program.

Alternately, doing the check after `main()` exits might be too late. Perhaps you have some objects that are known not to clean up properly at exit. You'd like to do the "at exit" check before those objects are destroyed (since while they're live, any memory they point to will not be considered a leak). In that case, you can call `HeapLeakChecker::NoGlobalLeaks()` manually, near the end of `main()`, and then call `HeapLeakChecker::CancelGlobalCheck()` to turn off the automatic post-`main()` check.

Finally, there's a helper macro for "strict" and "draconian" modes, which require all global memory to be freed before program exit. This freeing can be time-consuming and is often unnecessary, since libc cleans up all memory at program-exit for you. If you want the benefits of "strict"/"draconian" modes without the cost of all that freeing, look at `REGISTER_HEAPCHECK_CLEANUP` (in `heap-checker.h`). This macro allows you to mark specific cleanup code as active only when the heap-checker is turned on.

Explicit (Partial-program) Heap Leak Checking

Instead of whole-program checking, you can check certain parts of your code to verify they do not have memory leaks. This check verifies that between two parts of a program, no memory is allocated without being freed.

To use this kind of checking code, bracket the code you want checked by creating a `HeapLeakChecker` object at the beginning of the code segment, and call `NoLeaks()` at the end. These functions, and all others referred to in this file, are declared in `<gperftools/heap-checker.h>`.

Here's an example:

```
HeapLeakChecker heap_checker("test_foo");
{
  code that exercises some foo functionality;
  this code should not leak memory;
```

```
}
if (!heap_checker.NoLeaks()) assert(NULL == "heap memory leak");
```

Note that adding in the `HeapLeakChecker` object merely instruments the code for leak-checking. To actually turn on this leak-checking on a particular run of the executable, you must still run with the heap-checker turned on:

```
% env HEAPCHECK=local /usr/local/bin/my_binary_compiled_with_tcmalloc
```

If you want to do whole-program leak checking in addition to this manual leak checking, you can run in `normal` or some other mode instead: they'll run the "local" checks in addition to the whole-program check.

Disabling Heap-checking of Known Leaks

Sometimes your code has leaks that you know about and are willing to accept. You would like the heap checker to ignore them when checking your program. You can do this by bracketing the code in question with an appropriate heap-checking construct:

```
...
{
  HeapLeakChecker::Disabler disabler;
  <leaky code>
}
...
```

Any objects allocated by `leaky code` (including inside any routines called by `leaky code`) and any objects reachable from such objects are not reported as leaks.

Alternately, you can use `IgnoreObject()`, which takes a pointer to an object to ignore. That memory, and everything reachable from it (by following pointers), is ignored for the purposes of leak checking. You can call `UnIgnoreObject()` to undo the effects of `IgnoreObject()`.

Tuning the Heap Checker

The heap leak checker has many options, some that trade off running time and accuracy, and others that increase the sensitivity at the risk of returning false positives. For most uses, the range covered by the heap-check flavors is enough, but in specialized cases more control can be helpful.

These options are specified via environment varaiables.

This first set of options controls sensitivity and accuracy. These options are ignored unless you run the heap checker in as-is mode.

`HEAP_CHECK_AFTER_DESTRUCTORS`	Default: false	When true, do the final leak check after all other global

		destructors have run. When false, do it after all `REGISTER_HEAPCHECK_CLEANUP`, typically much earlier in the global-destructor process.
`HEAP_CHECK_IGNORE_THREAD_LIVE`	Default: true	If true, ignore objects reachable from thread stacks and registers (that is, do not report them as leaks).
`HEAP_CHECK_IGNORE_GLOBAL_LIVE`	Default: true	If true, ignore objects reachable from global variables and data (that is, do not report them as leaks).

These options modify the behavior of whole-program leak checking.

`HEAP_CHECK_MAX_LEAKS`	Default: 20	The maximum number of leaks to be printed to stderr (all leaks are still emitted to file output for pprof to visualize). If negative or zero, print all the leaks found.

These options apply to all types of leak checking.

`HEAP_CHECK_IDENTIFY_LEAKS`	Default: false	If true, generate the addresses of the leaked objects in the generated memory leak profile files.
`HEAP_CHECK_TEST_POINTER_ALIGNMENT`	Default: false	If true, check all leaks to see if they might be due to the use of unaligned pointers.
`HEAP_CHECK_POINTER_SOURCE_ALIGNMENT`	Default: sizeof(void*)	Alignment at which all pointers in memory are supposed to be located. Use 1 if

		any alignment is ok.
`PPROF_PATH`	Default: pprof	The location of the `pprof` executable.
`HEAP_CHECK_DUMP_DIRECTORY`	Default: /tmp	Where the heap-profile files are kept while the program is running.

Tips for Handling Detected Leaks

What do you do when the heap leak checker detects a memory leak? First, you should run the reported `pprof` command; hopefully, that is enough to track down the location where the leak occurs.

If the leak is a real leak, you should fix it!

If you are sure that the reported leaks are not dangerous and there is no good way to fix them, then you can use `HeapLeakChecker::Disabler` and/or `HeapLeakChecker::IgnoreObject()` to disable heap-checking for certain parts of the codebase.

In "strict" or "draconian" mode, leaks may be due to incomplete cleanup in the destructors of global variables. If you don't wish to augment the cleanup routines, but still want to run in "strict" or "draconian" mode, consider using `REGISTER_HEAPCHECK_CLEANUP`.

Hints for Debugging Detected Leaks

Sometimes it can be useful to not only know the exact code that allocates the leaked objects, but also the addresses of the leaked objects. Combining this e.g. with additional logging in the program one can then track which subset of the allocations made at a certain spot in the code are leaked.
To get the addresses of all leaked objects define the environment variable `HEAP_CHECK_IDENTIFY_LEAKS` to be `1`. The object addresses will be reported in the form of addresses of fake immediate callers of the memory allocation routines. Note that the performance of doing leak-checking in this mode can be noticeably worse than the default mode.

One relatively common class of leaks that don't look real is the case of multiple initialization. In such cases the reported leaks are typically things that are linked from some global objects, which are initialized and say never modified again. The non-obvious cause of the leak is frequently the fact that the initialization code for these objects

executes more than once.
E.g. if the code of some `.cc` file is made to be included twice into the binary, then the constructors for global objects defined in that file will execute twice thus leaking the things allocated on the first run.
Similar problems can occur if object initialization is done more explicitly e.g. on demand by a slightly buggy code that does not always ensure only-once initialization.

A more rare but even more puzzling problem can be use of not properly aligned pointers (maybe inside of not properly aligned objects). Normally such pointers are not followed by the leak checker, hence the objects reachable only via such pointers are reported as leaks. If you suspect this case define the environment variable `HEAP_CHECK_TEST_POINTER_ALIGNMENT` to be `1` and then look closely at the generated leak report messages.

How It Works

When a `HeapLeakChecker` object is constructed, it dumps a memory-usage profile named `<prefix>.<name>-beg.heap` to a temporary directory. When `NoLeaks()` is called (for whole-program checking, this happens automatically at program-exit), it dumps another profile, named `<prefix>.<name>-end.heap`. (`<prefix>` is typically determined automatically, and `<name>` is typically `argv[0]`.) It then compares the two profiles. If the second profile shows more memory use than the first, the `NoLeaks()` function will return false. For "whole program" profiling, this will cause the executable to abort (via `exit(1)`). In all cases, it will print a message on how to process the dumped profiles to locate leaks.

Detecting Live Objects

At any point during a program's execution, all memory that is accessible at that time is considered "live." This includes global variables, and also any memory that is reachable by following pointers from a global variable. It also includes all memory reachable from the current stack frame and from current CPU registers (this captures local variables). Finally, it includes the thread equivalents of these: thread-local storage and thread heaps, memory reachable from thread-local storage and thread heaps, and memory reachable from thread CPU registers.

In all modes except "draconian," live memory is not considered to be a leak. We detect this by doing a liveness flood, traversing pointers to heap objects starting from some initial memory regions we know to potentially contain live pointer data. Note that this flood might potentially not find some (global) live data region to start the flood from. If you find such, please file a bug.

The liveness flood attempts to treat any properly aligned byte sequences as pointers to heap objects and thinks that it found a good pointer whenever the current heap memory map contains an object with the address whose byte representation we found. Some pointers into not-at-start of object will also work here.

As a result of this simple approach, it's possible (though unlikely) for the flood to be inexact and occasionally result in leaked objects being erroneously determined to be live. For instance, random bit patterns can happen to look like pointers to leaked heap objects. More likely, stale pointer data not corresponding to any live program variables can be still present in memory regions, especially in thread stacks. For instance, depending on how the local `malloc` is implemented, it may reuse a heap object address:

```
char* p = new char[1];   // new might return 0x80000000, say.
delete p;
new char[1];             // new might return 0x80000000 again
// This last new is a leak, but doesn't seem it: p looks like it points to it
```

In other words, imprecisions in the liveness flood mean that for any heap leak check we might miss some memory leaks. This means that for local leak checks, we might report a memory leak in the local area, even though the leak actually happened before the `HeapLeakChecker` object was constructed. Note that for whole-program checks, a leak report *does* always correspond to a real leak (since there's no "before" to have created a false-live object).

While this liveness flood approach is not very portable and not 100% accurate, it works in most cases and saves us from writing a lot of explicit clean up code and other hassles when dealing with thread data.

Visualizing Leak with `pprof`

The heap checker automatically prints basic leak info with stack traces of leaked objects' allocation sites, as well as a pprof command line that can be used to visualize the call-graph involved in these allocations. The latter can be much more useful for a human to see where/why the leaks happened, especially if the leaks are numerous.

Leak-checking and Threads

At the time of HeapLeakChecker's construction and during `NoLeaks()` calls, we grab a lock and then pause all other threads so other threads do not interfere with recording or analyzing the state of the heap.

In general, leak checking works correctly in the presence of threads. However, thread stack data liveness determination (via `base/thread_lister.h`) does not work when the program is running under GDB, because the ptrace functionality needed for finding threads is already hooked to by GDB. Conversely, leak checker's ptrace

attempts might also interfere with GDB. As a result, GDB can result in potentially false leak reports. For this reason, the heap-checker turns itself off when running under GDB.

Also, `thread_lister` only works for Linux pthreads; leak checking is unlikely to handle other thread implementations correctly.

As mentioned in the discussion of liveness flooding, thread-stack liveness determination might mis-classify as reachable objects that very recently became unreachable (leaked). This can happen when the pointers to now-logically-unreachable objects are present in the active thread stack frame. In other words, trivial code like the following might not produce the expected leak checking outcome depending on how the compiled code works with the stack:

```
int* foo = new int [20];
HeapLeakChecker check("a_check");
foo = NULL;
// May fail to trigger.
if (!heap_checker.NoLeaks()) assert(NULL == "heap memory leak");
```

Maxim Lifantsev
Last modified: Fri Jul 13 13:14:33 PDT 2007