Where'd All That RAM Go?

Measuring Objects With Ownership-Aware Memory Profiling in C++

ALECTO IRENE PEREZ

Introduction

- My name is Alecto
- I'm a Senior Software Engineer at Vola Dynamics, The Volatility Company
- I wrote a memory profiler called `mem_profile`
- This is a personal project
- https://github.com/codeinred/mem_profile

Overview

This presentation will cover:

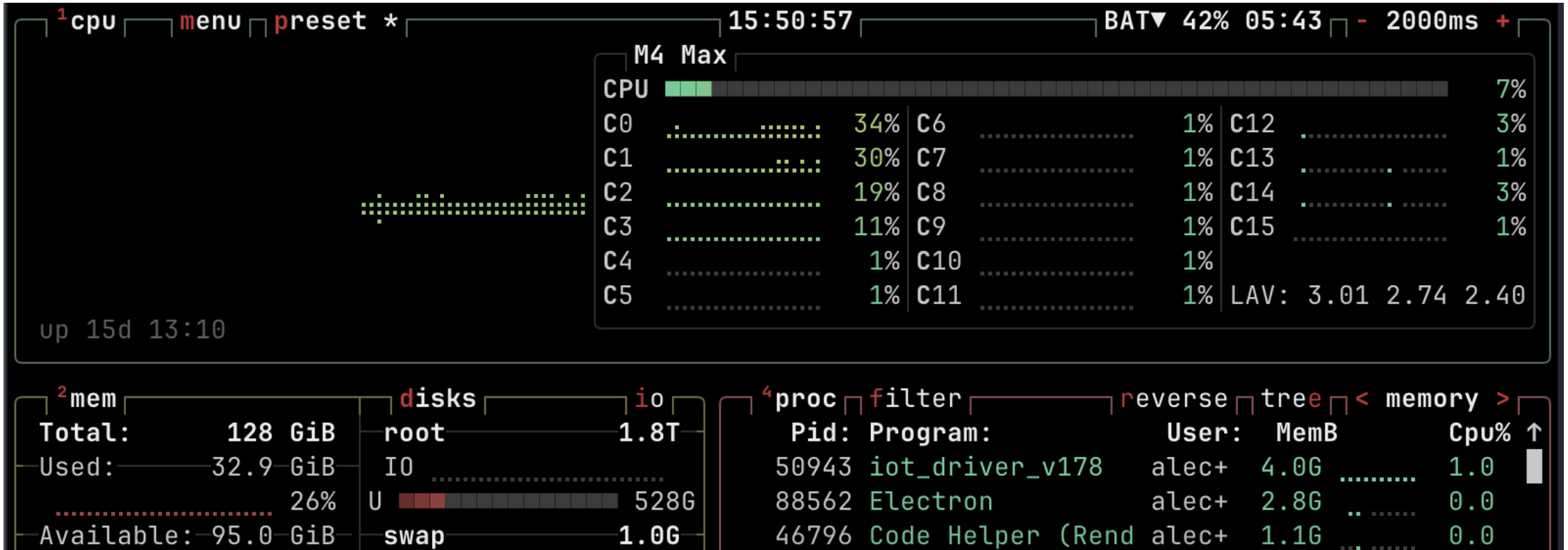
- Why we care about memory profiling (+background)
- What `mem_profile` supports (direct object-level memory statistics)
- How all of this works

```
❯ uv run python -m mp_reader type_stats malloc_stats.json --type 'my_object'

// Totals for my_object sizeof=72 bytes
// └── 398 bytes across 6 allocs and 1 instance
struct my_object
{
  bytes      a; // 10 bytes across 1 allocs : std::vector<std::byte>
  bytes      b; // 100 bytes across 1 allocs : std::vector<std::byte>
  string_map m; // 288 bytes across 4 allocs : std::map<std::string, std::string>
};
```

Why do we care about memory profiling?

- Memory on a system is a finite resource
- Swap is sometimes available, but it is orders of magnitude slower than RAM
- Out-of-memory is often unrecoverable



Why do we care about memory profiling?

- You can buy more RAM, you can get bigger servers...
- These things are *expensive*.
- If you ship to clients, they may not be able to get more RAM!
- **You need to be able to measure what you optimize.**

Measuring Memory Usage - An Introduction

Measuring Memory Usage

The simplest way to measure memory usage is with built-in utilities such as `time`.

Consider this program:

```
#include <cstdint>
#include <map>

int main() {
    std::map<int64_t, int64_t> m;
    for (size_t i = 0; i < 1'000'000; i++) {
        m[i] = i * i;
    }
}
```

Measuring Memory Usage

`time` can measure peak memory footprint with either `-l` or `-v`:

```
❯ \time -l ./a.out
        0.06 real         0.06 user         0.00 sys
            49332224  maximum resident set size
                   0  average shared memory size
                   0  average unshared data size
                   0  average unshared stack size
                3210  page reclaims
                   1  page faults
                   0  swaps
                   0  block input operations
                   0  block output operations
                   0  messages sent
                   0  messages received
                   0  signals received
                   0  voluntary context switches
                   3  involuntary context switches
           816149941  instructions retired
           237223462  cycles elapsed
            49217896  peak memory footprint
```

Measuring Memory Usage - `time`

Pros:

- Quick
- Easy
- You get one number: peak memory footprint

Cons:

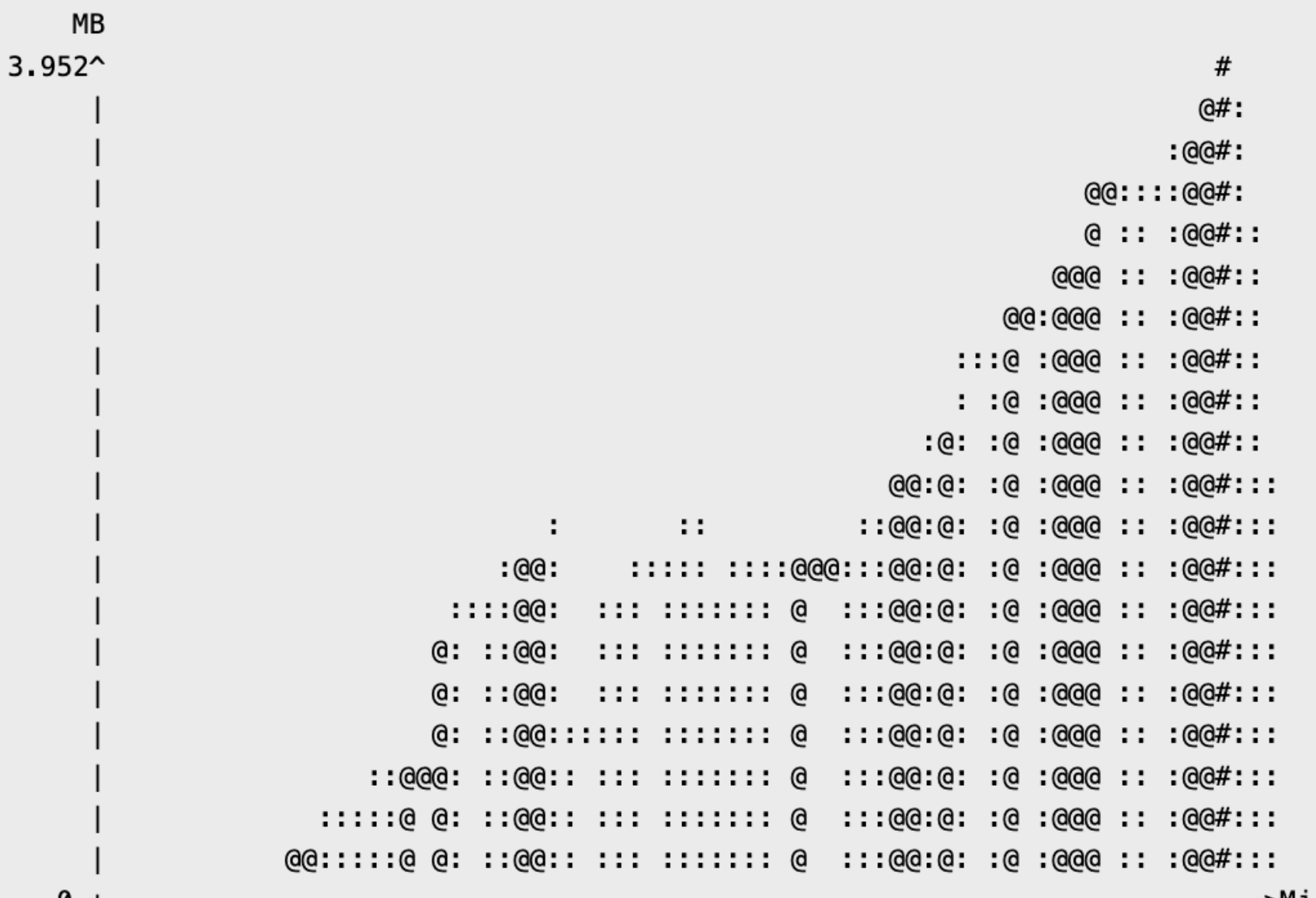
- No visibility into what was using memory
- You only get one number: peak memory footprint

Measuring Memory Usage - `valgrind massif`

- `massif` can measure memory usage over time
- You get more granularity with regards to when peak memory usage occurs
- Can detect leaks!

9. Massif: a heap profiler

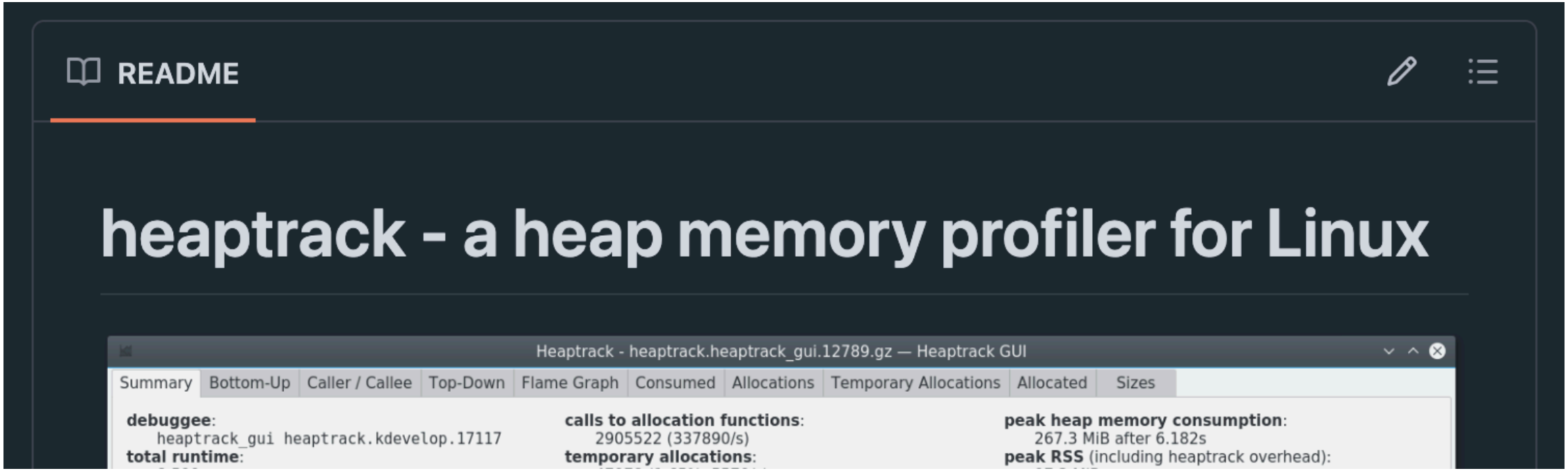
Table of Contents

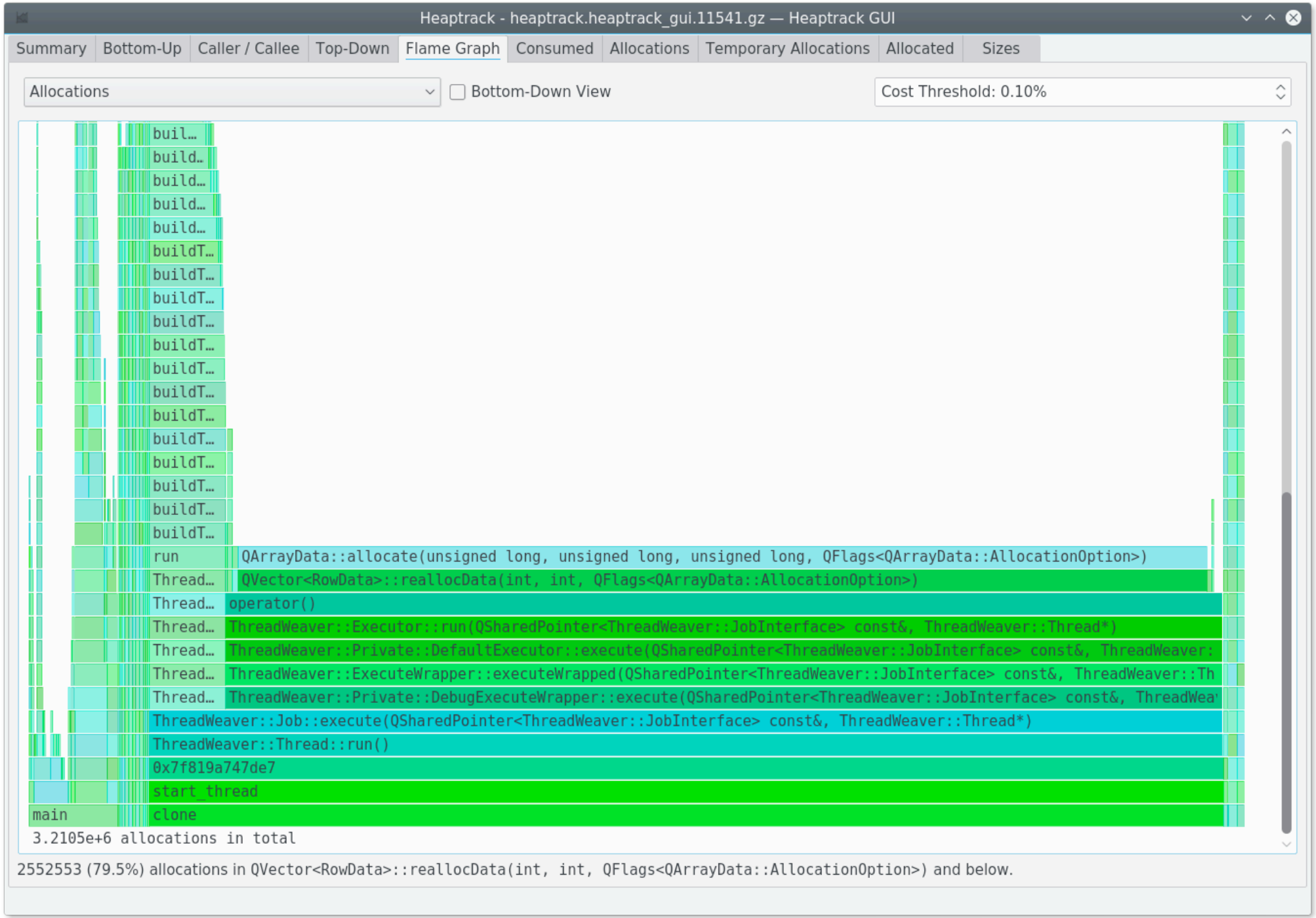



```
Number of snapshots: 63
 Detailed snapshots: [3, 4, 10, 11, 15, 16, 29, 33, 34, 36, 39, 41,
                      42, 43, 44, 49, 50, 51, 53, 55, 56, 57 (peak)]
```

Measuring Memory Usage - KDE `heaptrack`

- Can see where allocations occur
- Can get flamegraph of memory allocations
- Can identify temporary allocations




Heaptrack - heaptrack.heaptrack_gui.11541.gz — Heaptrack GUI
Summary
Bottom-Up
Caller / Callee
Top-Down
Flame Graph
Consumed
Allocations
Temporary Allocations
Allocated
Sizes
Allocations
Bottom-Down View
Cost Threshold: 0.10%
buil…
build…
build…
build…
build…
buildT…
buildT…
buildT…
buildT…
buildT…
buildT…
buildT…
buildT…
buildT…
buildT…
buildT…
buildT…
run
QArrayData::allocate(unsigned long, unsigned long, unsigned long, QFlags<QArrayData::AllocationOption>)
Thread…
QVector<RowData>::reallocData(int, int, QFlags<QArrayData::AllocationOption>)
Thread…
operator()
Thread…
ThreadWeaver::Executor::run(QSharedPointer<ThreadWeaver::JobInterface> const&, ThreadWeaver::Thread*)
Thread…
ThreadWeaver::Private::DefaultExecutor::execute(QSharedPointer<ThreadWeaver::JobInterface> const&, ThreadWeaver:
Thread…
ThreadWeaver::ExecuteWrapper::executeWrapped(QSharedPointer<ThreadWeaver::JobInterface> const&, ThreadWeaver::Th
Thread…
ThreadWeaver::Private::DebugExecuteWrapper::execute(QSharedPointer<ThreadWeaver::JobInterface> const&, ThreadWea
ThreadWeaver::Job::execute(QSharedPointer<ThreadWeaver::JobInterface> const&, ThreadWeaver::Thread*)
ThreadWeaver::Thread::run()
0x7f819a747de7
start_thread
main
clone
3.2105e+6 allocations in total
2552553 (79.5%) allocations in QVector<RowData>::reallocData(int, int, QFlags<QArrayData::AllocationOption>) and below.

So why another profiler?

So why another profiler?

We write code in terms of data structures, containers, and objects. I want to be able to measure memory in terms of these things!

- Which objects are consuming memory?
- Why isn't memory released?
- What members/subobjects are using that memory?

`mem_profile` can answer these questions!

We resolved limitations of similar efforts

mem_profile's RAII-based approach resolves many limitations faced by prior attempts at profiling via object introspection.

- `mem_profile` can profile user-defined containers
- `mem_profile` handles template specialization of containers
- `mem_profile` handles virtual inheritance
- `mem_profile` handles c-style unions, AND ALSO subobjects allocated with placement-new.
- `mem_profile` resolves shared ownership
- if a pointer is freed in the destructor, `mem_profile` handles it

Can Your Profiler Do This?

mem_profile embeds type and layout information in a binary to provide comprehensive statistics, even for release builds.

```cpp
using bytes      = std::vector<std::byte>;
using string_map = std::map<std::string, std::string>;

struct my_object {
    bytes      a;
    bytes      b;
    string_map m;
};

int main() {
    my_object m{
        bytes(10),
        bytes(100),
        string_map{
            {"key1", "Hello, world"},
            {"key2", "The quick brown fox jumps over the lazy dogs"},
            {"key3", "Have a great CppCon!"},
        },
    };
}
```

```
 mem_profile/stats_reader on  main at v0.1.0 via  v3.13.2
❯ uv run python -m mp_reader type_stats test_files/macos/objects.json \
    --type 'my_object'

// Totals for my_object sizeof=72 bytes
//  └── 398 bytes across 6 allocs and 1 instance
struct my_object
{
  bytes      a; // 10 bytes across 1 allocs : std::vector<std::byte>
  bytes      b; // 100 bytes across 1 allocs : std::vector<std::byte>
  string_map m; // 288 bytes across 4 allocs : std::map<std::string, std::string>
};

 mem_profile/stats_reader on  main at v0.1.0 via  v3.13.2
❯
```

```
 mem_profile/stats_reader on  main at v0.1.0 via  v3.13.2
❯ uv run python -m mp_reader type_stats test_files/macos/objects.json \
    --type 'my_object' --show-offsets

// Totals for my_object sizeof=72 bytes
// └── 398 bytes across 6 allocs and 1 instance
struct my_object
{
  bytes      a; //    0..  24   10 bytes across 1 allocs : std::vector<std::byte>
  bytes      b; //   24..  48   100 bytes across 1 allocs : std::vector<std::byte>
  string_map m; //   48..  72   288 bytes across 4 allocs : std::map<std::string, std::string>
};

 mem_profile/stats_reader on  main at v0.1.0 via  v3.13.2
❯
```

examples > src > objects_in_lambda.cpp > ...

```cpp
#include <cstddef>
#include <vector>

using bytes   = std::vector<std::byte>;
using floats  = std::vector<float>;
using doubles = std::vector<double>;

int main() {
    auto myLambda = [a = bytes(10),   //
                     b = floats(100), //
                     c = doubles(1000)]() {};
}

```

```
 mem_profile/stats_reader on  main at v0.1.0 via  v3.13.2
❯ uv run python -m mp_reader type_stats test_files/macos/objects_in_lambda.json --type 'lambda'

// Totals for (lambda at /opt/pages/mem_profile/examples/src/objects_in_lambda.cpp:9:21) sizeof=72 bytes
// └── 8,410 bytes across 3 allocs and 1 instance
struct (lambda at /opt/pages/mem_profile/examples/src/objects_in_lambda.cpp:9:21)
{
  bytes   (unnamed); // 10 bytes across 1 allocs : std::vector<std::byte>
  floats  (unnamed); // 400 bytes across 1 allocs : std::vector<float>
  doubles (unnamed); // 8,000 bytes across 1 allocs : std::vector<double>
};

 mem_profile/stats_reader on  main at v0.1.0 via  v3.13.2
❯
```

```cpp
#include <cstddef>
#include <vector>

using bytes = std::vector<std::byte>;

struct array3 {
    bytes elems[3];
};

int main() {
    array3 values{
        bytes(10),
        bytes(100),
        bytes(1000),
    };
}
```

```
 mem_profile/stats_reader on  main +26 -10 !2 at v0.1.0 via  v3.13.2
❯ uv run python -m mp_reader type_stats test_files/macos/objects_in_array.json --type 'array3'

// Totals for array3 sizeof=72 bytes
// └── 1,110 bytes across 3 allocs and 1 instance
struct array3
{
  bytes[3] elems; // [0] 10 bytes across 1 allocs : std::vector<std::byte>
                  // [1] 100 bytes across 1 allocs : std::vector<std::byte>
                  // [2] 1,000 bytes across 1 allocs : std::vector<std::byte>
};

 mem_profile/stats_reader on  main +26 -10 !2 at v0.1.0 via  v3.13.2
❯
```

```cpp
template <class T, size_t Cap>
struct svector {
    size_t count = 0;
    alignas(T) std::byte buffer[sizeof(T) * Cap];

    void push_back(T&& value) {…
    }
    T* data() noexcept { return reinterpret_cast<T*>(buffer); }
    ~svector() {
        for (size_t i = 0; i < count; i++) data()[i].~T();
    }
};

int main() {
    svector<bytes, 10> v;
    v.push_back(bytes(10));    // [0]
    v.push_back(bytes(100));   // [1]
    v.push_back(bytes(1000));  // [2]
    v.push_back(bytes());      // [3] - empty: no allocation will be recorded here
    v.push_back(bytes(12345)); // [4]
}
```

```
 mem_profile/stats_reader on  main at v0.1.0 via  v3.13.2
❯ uv run python -m mp_reader type_stats test_files/macos/objects_in_svector.json --type 'svector'

// Totals for svector<std::vector<std::byte>, 10> sizeof=248 bytes
//  └── 13,455 bytes across 4 allocs and 1 instance
struct svector<std::vector<std::byte>, 10>
{
  size_t          count;
  std::byte[240] buffer; // [0] 10 bytes across 1 allocs : std::vector<std::byte>
                         // [1] 100 bytes across 1 allocs : std::vector<std::byte>
                         // [2] 1,000 bytes across 1 allocs : std::vector<std::byte>
                         // [4] 12,345 bytes across 1 allocs : std::vector<std::byte>
};

 mem_profile/stats_reader on  main +4 -9 !1 at v0.1.0 via  v3.13.2
❯
```

examples > src > objects_in_svector_martinus.cpp > ...

```cpp
#include <ankerl/svector.h>      You, yesterday • Add examples 'objects_in_svector' a
#include <cstddef>
#include <vector>

using bytes = std::vector<std::byte>;


int main() {
    ankerl::svector<bytes, 16> sv;
    sv.push_back(bytes(10));
    sv.push_back(bytes(100));
    sv.push_back(bytes(1000));
}

```

```
 mem_profile/stats_reader on  main +26 -10 !2 at v0.1.0 via  v3.13.2
❯ uv run python -m mp_reader type_stats test_files/macos/objects_in_svector_martinus.json \
    --type 'svector' \
    --replace 'detail::size_of_svector<vector<byte, allocator<byte>>>(16UL)' \
    --replace-with '...'

// Totals for ankerl::svector<std::vector<std::byte>, 16> sizeof=392 bytes
// └── 1,110 bytes across 3 allocs and 1 instance
struct ankerl::svector<std::vector<std::byte>, 16>
{
  std::array<uint8_t, ...> m_data; // +8 10 bytes across 1 allocs : std::vector<std::byte>
                                   // +32 100 bytes across 1 allocs : std::vector<std::byte>
                                   // +56 1,000 bytes across 1 allocs : std::vector<std::byte>
};

 mem_profile/stats_reader on  main +26 -10 !2 at v0.1.0 via  v3.13.2
❯
```

```cpp
#include <cstddef>      You, 2 days ago • Add more examples
#include <vector>

using bytes = std::vector<std::byte>;


int main() {
    std::vector values{
        bytes(10),
        bytes(100),
        bytes(1000),
    };
}

```

```
 mem_profile/stats_reader on  main at v0.1.0 via  v3.13.2
❯ uv run python -m mp_reader type_stats test_files/macos/objects_in_vector.json --type 'std::vector' \
    --clean-members

// Totals for std::vector<std::byte> sizeof=24 bytes
// └── 2,220 bytes across 6 allocs and 6 instances
struct std::vector<std::byte>
{
  // (6 non-owning fields cleaned)

  ~std::vector<std::byte>();
  // directly owned (or unannotated):
  // └── 2,220 bytes across 6 allocs
};



// Totals for std::vector<std::vector<std::byte>> sizeof=24 bytes
// └── 1,182 bytes across 4 allocs and 1 instance
struct std::vector<std::vector<std::byte>>
{
  // (6 non-owning fields cleaned)

  ~std::vector<std::vector<std::byte>>();
  // directly owned (or unannotated):
  // └── 72 bytes across 1 allocs
  // children on heap:
  // └── 1,110 bytes across 3 allocs : std::vector<std::byte>
};
```

How big is an object?

Measuring Objects and the Perils of Shared Ownership

Measuring Objects - Simple Example

```
#include <memory>

struct A {
    std::shared_ptr<char[]> data;

    static A alloc( size_t bytes ) {
        return A{std::shared_ptr<char[]>(new char[bytes])};
    }
};

int main() {
    // How big is this object?
    A a = A::alloc(1'000'000);
}
```

Measuring objects - Simple Example

```
int main() {
    // How big is this object?
    A a = A::alloc(1'000'000);
}
```

- `A` is 16 bytes (assuming 8-byte pointers)
- `a.data` is holds 1,000,000 bytes, allocated on the heap

Therefore:

- `a` takes up 1,000,016 bytes total (1,000,000 on the heap)

Measuring Objects - Simple Example

Suppose we had a magic function, `measure_heap(object)`, that tells you how much memory is reachable from an object:

```
int main() {
    // How big is this object?
    A a = A::alloc(1'000'000);

    fmt::println("total size: {}", measure_heap(a));
}
```

This prints `total size: 1000000`, or 1,000,000 bytes.

Unfortunately, not everything is simple.

Measuring Objects - 💥

Sometimes, objects have shared custody.

Q: Is this correct? (output: `total bytes = 2000000`)

```
int main() {
    // How big is this object?
    A a = A::alloc(1'000'000);
    A b = A::alloc(1'000'000);

    size_t total_bytes = measure_heap(a) + measure_heap(b);

    fmt::println("total bytes = {}", total_bytes);
}
```

Measuring Objects - 💥

Sometimes, objects have shared custody.

Q: Is this correct? (output: `total bytes = 2000000`)

```
int main() {
    // How big is this object?
    A a = A::alloc(1'000'000);
    A b = A::alloc(1'000'000);

    size_t total_bytes = measure_heap(a) + measure_heap(b);

    fmt::println("total bytes = {}", total_bytes);
}
```

Yes, this is fine. `a` and `b` don't share any memory.

Measuring Objects - 💥

Sometimes, objects have shared custody.

Q: Is this correct? (output: `total bytes = 2000000`)

```
int main() {
    // How big is this object?
    A a = A::alloc(1'000'000);
    A b = a;

    size_t total_bytes = measure_heap(a) + measure_heap(b);

    fmt::println("total bytes = {}", total_bytes);
}
```

Measuring Objects - 💥

Sometimes, objects have shared custody.

Q: Is this correct? (output: `total bytes = 2000000`)

```
int main() {
    // How big is this object?
    A a = A::alloc(1'000'000);
    A b = a;

    size_t total_bytes = measure_heap(a) + measure_heap(b);

    fmt::println("total bytes = {}", total_bytes);
}
```

NO! This is not correct! Even with a magic `measure_heap` function, these objects share memory!

Measuring Objects - How do we fix this?

Measuring Objects - How do we fix this?

Let's take some simple code, and desugar it:

```
A a = A::alloc(1'000'000); // use count: 1
A b = a;                   // use count: 1 -> 2

// ... (stuff happens) ...

b.~A(); // b's lifetime ends (IMPLICIT)
        // use count: 2 -> 1
a.~A(); // a's lifetime ends (IMPLICIT)
        // use count: 1 -> 0 (deallocation occurs)
```

- `b` is destroyed first, and performs no deallocations
- `a` is destroyed last, and `a`'s destructor cleans up the memory.

Measuring Objects - How do we fix this?

- Ownership is shared, but deallocation is unique.
- In a leak-free program, any block of memory should be deallocated precisely once.
- In our example, *only* `a` 's destructor frees memory
- Therefore: **attribute the memory to `a` !**

```
int main() {
    // How big is this object?
    A a = A::alloc(1'000'000);
    A b = a;

    // ...
}
```

Measuring Objects - A Destructor-Based Approach

These are the rules:

- We attribute memory to the objects that actually deallocate that memory. `a` performed the deallocation, so the memory held by the shared pointer is deallocated to `a`
- Any memory deallocated in a destructor call was owned by the object being destroyed
- If no deallocations occur on destruction, there's nothing to attribute
- Because each deallocation is unique, this *also* correctly assigns an owner, *even when the memory is shared*

Is This Correct?

I claim that it is!

- This approach is *generic* - any object that owns memory *must free that memory*
- This approach ignores non-owning types - we don't have to care about `std::span` or `string_view`, because these don't deallocate!
- This approach handles both unique ownership and shared ownership
- Memory is *always* attributed to the *last remaining object* which owns a piece of memory.
- This also allows us to discover why memory wasn't freed sooner!

The Profiler

Let's Measure Some Objects!

Measuring Objects - Test Code

```
#include <memory> // shared.cpp, compiled as `build/shared`
#include <cstdio>

struct A {
    std::shared_ptr<char[]> data;

    static A alloc( size_t bytes ) {
        return A{std::shared_ptr<char[]>(new char[bytes])};
    }
};

int main() {
    A a = A::alloc(1'000'000);
    A b = a;
    printf("a.this = %p\n", &a);
    printf("b.this = %p\n", &b);
}
```

Measuring Objects - Compiling

You can build your project with `mem_profile` by adding these lines near the top of your `CMakeLists.txt`.

```
find_package(mem_profile REQUIRED)
link_libraries(mp::mp_build_with_plugin)
```

Use llvm `clang` when compiling:

```
env CC=clang CXX=clang++ cmake -B build -G Ninja
cmake --build build
```

Note: on macos, this should be `/opt/homebrew/opt/llvm/bin/clang++`

Measuring Objects - Compiling

- `mp_build_with_plugin` is an INTERFACE target
- Linking it adds compile flags to inject the plugin when compiling with clang.
- Use `link_libraries` to ensures the plugin is linked against all targets in the binary.
- The plugin injects information into destructor calls to enable introspection and ownership tracing

Compile Output with Printing Enabled

```
Rewrote my_object::~my_object @ /opt/pages/mem_profile/examples/src/objects.cpp:9:8
inline ~my_object() noexcept __noinline__ {
    static constexpr const char *__MP_FIELD_NAMES[3] = {"a", "b", "m"};
    static constexpr const char *__MP_FIELD_TYPES[3] = {"bytes", "bytes", "string_map"};
    static constexpr unsigned long __MP_FIELD_SIZES[3] = {24UL, 24UL, 24UL};
    static constexpr unsigned long __MP_FIELD_OFFSETS[3] = {0UL, 24UL, 48UL};
    static constexpr const char *__MP_BASE_TYPES[0] = {};
    static constexpr unsigned long __MP_BASE_SIZES[0] = {};
    static constexpr unsigned long __MP_BASE_OFFSETS[0] = {};
    static constexpr _mp_type_data __MP_TYPE_DATA = {sizeof(my_object), "my_object", 0UL,
BASE_SIZES, __MP_BASE_OFFSETS};
    save_state(this, __builtin_alloca(40UL), __MP_TYPE_DATA);
}
[2/2] Linking CXX executable objects
```

You can use `MEM_PROFILE_PRINT_BODY=1` *to view instrumented destructor calls.*

Measuring Objects - Running the Profiler

Let's profile our executable. We can use either `LD_PRELOAD` (on linux) or `DYLD_INSERT_LIBRARIES` to inject the profiler:

```
# linux
env LD_PRELOAD=/usr/local/lib/libmp_runtime.so build/shared

# macos
env DYLD_INSERT_LIBRARIES=/usr/local/lib/libmp_runtime.dylib build/shared
```

The profiler will be initialized on load. `libmp_runtime` provides definitions for `malloc`, `free`, `new`, `delete`, etc that record allocations and deallocations as they occur.

Measuring Objects - Output

When we run our executable with the profiler, it generates a file `malloc_stats.json`. This contains all of the information necessary for analysis.

```
$ env LD_PRELOAD=/usr/local/lib/libmp_runtime.so build/shared
a.this = 0x16d6fe980
b.this = 0x16d6fe970

$ ll malloc_stats.json
-rw-r--r--@ 1 alecto  staff   3.1K Sep 11 13:44 malloc_stats.json
```

You can configure the output filename by setting `MEM_PROFILE_OUT=<filename>`.

```
$ env MEM_PROFILE_OUT=output.json LD_PRELOAD=... /path/to/executable
```

Measuring Objects - Output

If we dump an event trace for the file, we can see that there are two free events:

- a 1MB allocation for the `char[]` array held by the shared pointer, and
- a 32-byte allocation for the metadata the shared pointer uses to track the object.

Note that the `this` pointer recorded in `malloc_stats` matches the address displayed by the executable.

```
#                  module    subcommand  input file
#                  v         v           v
$ uv run python -m mp_reader dump_events malloc_stats.json
```


type: FREE
addr: 128008000
size: 1000000
/opt/homebrew/Cellar/llvm/20.1.7/bin/../include/c++/v1/__memory/shared_count.h:121:7 (inlined)
std::__1::__shared_weak_count::__release_shared[abi:ne200100]()
/opt/homebrew/Cellar/llvm/20.1.7/bin/../include/c++/v1/__memory/shared_ptr.h:558:17
std::__1::shared_ptr<char []>::~shared_ptr[abi:ne200100]()
OBJECT this=0x16d6fe980 offset=0 size=16 type=std::shared_ptr<char[]>
/opt/pages/cppcon/example/shared.cpp:0:8
A::~A()
OBJECT this=0x16d6fe980 size=16 type=A
/opt/pages/cppcon/example/shared.cpp:17:1
main
type: FREE
addr: 125605f50
size: 32
/opt/homebrew/Cellar/llvm/20.1.7/bin/../include/c++/v1/__memory/shared_count.h:0:7 (inlined)
std::__1::__shared_weak_count::__release_shared[abi:ne200100]()
/opt/homebrew/Cellar/llvm/20.1.7/bin/../include/c++/v1/__memory/shared_ptr.h:558:17
std::__1::shared_ptr<char []>::~shared_ptr[abi:ne200100]()
OBJECT this=0x16d6fe980 offset=0 size=16 type=std::shared_ptr<char[]>
/opt/pages/cppcon/example/shared.cpp:0:8
A::~A()
OBJECT this=0x16d6fe980 size=16 type=A
/opt/pages/cppcon/example/shared.cpp:17:1
main

Output (for reference)

```
$ env LD_PRELOAD=/usr/local/lib/libmp_runtime.so build/shared
a.this = 0x16d6fe980
b.this = 0x16d6fe970
```

Measuring Objects - The Trace

- Each entry in the trace corresponds to a stack frame.
- Some entries correspond to destructor calls
- These destructor calls are annotated with information about the object being destroyed, namely,
  - the `this` pointer of the object,
  - it's size,
  - and it's typename.
- Additional type information is also recorded and stored in `malloc_stats.json`, although this is not displayed in the trace.

Measuring Objects - Statistics

We can use the `mp_reader stats` command to generate stats for the program.

These statistics are on a per-type basis.

```
❯ uv run python -m mp_reader stats /opt/3rd/FTXUI/malloc_stats.json
Allocation Statistics by Type:

     140 objects     1,438,080 bytes  std::vector<ftxui::Pixel>
      10 objects     1,338,720 bytes  ftxui::Screen
      10 objects     1,338,480 bytes  std::vector<std::vector<ftxui::Pixel>>
      10 objects     1,338,480 bytes  ftxui::Image
     600 objects       708,000 bytes  ftxui::flexbox_helper::Global
     600 objects       536,640 bytes  std::vector<ftxui::flexbox_helper::Block>
   1,930 objects       500,400 bytes  std::shared_ptr<ftxui::Node>
     290 objects       498,880 bytes  std::vector<std::shared_ptr<ftxui::Node>>
      40 objects       498,880 bytes  ftxui::(anonymous namespace)::Border
     100 objects       495,840 bytes  ftxui::(anonymous namespace)::Flexbox
     110 objects       442,200 bytes  ftxui::(anonymous namespace)::VBox
   2,950 objects       307,200 bytes  std::vector<ftxui::box_helper::Element>
```

Measuring Objects - Peak Usage

By default, mem_profile shows accumulated stats over the full runtime of the program, but you can limit statistics to objects alive at peak usage.

```
❯ uv run python -m mp_reader stats /opt/3rd/FTXUI/malloc_stats.json --filter-peak --count 6
Allocation Statistics by Type:

     14 objects       143,808 bytes  std::vector<ftxui::Pixel>
      1 object        133,848 bytes  std::vector<std::vector<ftxui::Pixel>>
      1 object        133,848 bytes  ftxui::Image
      1 object        133,848 bytes  ftxui::Screen
    193 objects        38,240 bytes  std::shared_ptr<ftxui::Node>
     29 objects        38,088 bytes  std::vector<std::shared_ptr<ftxui::Node>>
                              ...
              10 entries filtered

  9,640 allocs              21 bytes  <untyped>

    302 objects       194,216 bytes  <total>
```

Measuring Objects - Breakdown by Member

Stats by type are nice, but we can do a lot more!

- We know the `this` pointer of every object
- We know the size of every object
- *this allows us to compute the offset of subobjects within a greater whole!*

Additionally:

- We already have a compiler plugin
- The compiler knows every field in an object, it's name, it's type, and it's offset
- We can use this to embed any and all desired type info, directly in the binary!

```
❯ uv run python -m mp_reader type_stats /opt/3rd/FTXUI/malloc_stats.json \
  --type '^ftxui' --filter-mode=REGEX --clean-members

// Totals for ftxui::Screen sizeof=128 bytes
// └── 1,338,720 bytes across 150 allocs and 10 instances
struct ftxui::Screen
  : Image // 1,338,480 bytes across 140 allocs : ftxui::Image
{
  // (2 non-owning fields cleaned)
  std::vector<std::string> hyperlinks_; // 240 bytes across 10 allocs : std::vector<std::string>
};


// Totals for ftxui::Image sizeof=56 bytes
// └── 1,338,480 bytes across 140 allocs and 10 instances
struct ftxui::Image
{
  // (3 non-owning fields cleaned)
  std::vector<std::vector<Pixel>> pixels_; // 1,338,480 bytes across 140 allocs : std::vector<std::vector<ftxui::P
};


// Totals for ftxui::flexbox_helper::Global sizeof=88 bytes
// └── 708,000 bytes across 2100 allocs and 600 instances
struct ftxui::flexbox_helper::Global
{
  // (3 non-owning fields cleaned)
  std::vector<Block> blocks; // 536,640 bytes across 600 allocs : std::vector<ftxui::flexbox_helper::Block>
  std::vector<Line>  lines; // 171,360 bytes across 1,500 allocs : std::vector<ftxui::flexbox_helper::Line>
};
```

Testing The Profiler on a Non-Trivial Codebase

We're going to profile CMake

Profiling CMake

- 935 translation units, a million lines of code, multiple dependencies…
- We can build the *whole thing* with the compiler plugin, and profile it with `mem_profile`

```
// Totals for cmState sizeof=600 bytes
// └── 3,474,745 bytes across 36738 allocs and 3 instances
struct cmState
{
  cmPropertyDefinitionMap                                 PropertyDefinitions; // 2,592 bytes across 18 allocs : cmPropertyDefinitionMap
  std::vector<std::string>                                EnabledLanguages; // 144 bytes across 3 allocs : std::vector<std::string>
  std::unordered_map<std::string, Command>                BuiltinCommands; // 34,569 bytes across 468 allocs : std::unordered_map<std::string, std::function
<bool (const std::vector<cmListFileArgument> &, cmExecutionStatus &)>>
  std::unordered_map<std::string, Command>                ScriptedCommands; // 1,973,623 bytes across 18,180 allocs : std::unordered_map<std::string, std::f
unction<bool (const std::vector<cmListFileArgument> &, cmExecutionStatus &)>>
  std::unordered_set<std::string>                         FlowControlCommands; // 2,592 bytes across 54 allocs : std::unordered_set<std::string>
  cmPropertyMap                                           GlobalProperties; // 3,049 bytes across 55 allocs : cmPropertyMap
  std::unique_ptr<cmCacheManager>                         CacheManager; // 78,980 bytes across 1,207 allocs : std::unique_ptr<cmCacheManager>
  std::unique_ptr<cmGlobVerificationManager>              GlobVerificationManager; // 216 bytes across 3 allocs : std::unique_ptr<cmGlobVerificationManager>
  cmLinkedTree<cmStateDetail::BuildsystemDirectoryStateType> BuildsystemDirectory; // 8,192 bytes across 43 allocs : cmLinkedTree<cmStateDetail::BuildsystemDir
ectoryStateType>
  cmLinkedTree<std::string>                               ExecutionListFiles; // 40,088 bytes across 315 allocs : cmLinkedTree<std::string>
  cmLinkedTree<cmStateDetail::PolicyStackEntry>           PolicyStack; // 3,072 bytes across 6 allocs : cmLinkedTree<cmStateDetail::PolicyStackEntry>
  cmLinkedTree<cmStateDetail::SnapshotDataType>           SnapshotData; // 129,024 bytes across 6 allocs : cmLinkedTree<cmStateDetail::SnapshotDataType>
  cmLinkedTree<cmDefinitions>                             VarTree; // 1,198,188 bytes across 16,374 allocs : cmLinkedTree<cmDefinitions>
  std::string                                             SourceDirectory;
  std::string                                             BinaryDirectory;
  bool                                                    IsGeneratorMultiConfig;
  bool                                                    WindowsShell;
  bool                                                    WindowsVSIDE;
  bool                                                    CbsMultiIDE;
```

```
 cmake on  master +25 -3 !3 ?5 ⇡1 via  v4.1.1
❯ ll
total 99941168
drwxr-xr-x@   7 alecto  staff   224B Aug 28 20:09 Auxiliary
-rwxr-xr-x@   1 alecto  staff    65K Aug 28 20:09 bootstrap
drwxr-xr-x@  27 alecto  staff   864B Sep 11 10:06 build
drwxr-xr-x@   4 alecto  staff   128B Sep  1 21:54 build_mp
-rw-r--r--@   1 alecto  staff    15G Aug 28 22:20 cmake_stats.json
-rw-r--r--@   1 alecto  staff   794B Aug 28 20:09 cmake_uninstall.cmake.in
-rw-r--r--@   1 alecto  staff   9.8K Aug 28 20:09 CMakeCPack.cmake
-rw-r--r--@   1 alecto  staff    12K Aug 28 20:09 CMakeCPackOptions.cmake.in
-rw-r--r--@   1 alecto  staff   153B Aug 28 20:09 CMakeGraphVizOptions.cmake
-rw-r--r--@   1 alecto  staff    22K Aug 28 20:19 CMakeLists.txt
-rw-r--r--@   1 alecto  staff   4.4K Aug 28 20:09 CMakeLogo.gif
-rw-r--r--@   1 alecto  staff   5.2K Aug 28 20:09 CompileFlags.cmake
-rwxr-xr-x@   1 alecto  staff    99B Aug 28 20:09 configure
-rw-r--r--@   1 alecto  staff   3.2K Aug 28 20:09 CONTRIBUTING.rst
-rw-r--r--@   1 alecto  staff   4.1K Aug 28 20:09 CONTRIBUTORS.rst
-rw-r--r--@   1 alecto  staff   564B Aug 28 20:09 CTestConfig.cmake
-rw-r--r--@   1 alecto  staff   7.7K Aug 28 20:09 CTestCustom.cmake.in
-rw-r--r--@   1 alecto  staff   372B Aug 28 20:09 DartConfig.cmake
-rw-r--r--@   1 alecto  staff    31G Aug 29 14:08 event_table.json
```

```
❯ uv run python -m mp_reader stats --top-n-layouts=100 /opt/3rd/CMake/cmake_stats.json
  29m 6.0s to load json...
  7m 55.0s to create output record...
Allocation Statistics by Type:

  14,215 objects   544,249,813 bytes  std::vector<char>
   5,719 objects   375,678,464 bytes  std::unique_ptr<cmUVStreamReadHandle>
   5,719 objects   375,175,192 bytes  cmUVStreamReadHandle
 116,042 objects   202,228,957 bytes  std::vector<cmListFileFunction>
 764,374 objects   180,782,606 bytes  std::shared_ptr<const cmListFileFunction::Implementation>
 764,374 objects   180,782,606 bytes  cmListFileFunction
2,109,345 objects   139,738,246 bytes  cmListFileContext
     326 objects   137,039,317 bytes  std::unique_ptr<cmState>
 660,459 objects   104,075,801 bytes  std::vector<cmListFileArgument>
 780,757 objects    96,498,522 bytes  std::vector<std::string>
 575,310 objects    95,172,718 bytes  cmListFileFunction::Implementation
  20,832 objects    90,850,328 bytes  std::__function::__value_func<bool (const std::vector<cmListFileArgument> &, cmExecutionStatus &)>
  20,832 objects    90,850,328 bytes  std::function<bool (const std::vector<cmListFileArgument> &, cmExecutionStatus &)>
   9,206 objects    88,932,201 bytes  cmListFile
     652 objects    87,755,955 bytes  std::__hash_table<std::__hash_value_type<std::string, std::function<bool (const std::vector<cmListFileArgument> &, cmExecutionStatus &)>>, std::__unordered_map_hasher<std::string, std::__hash_value_type<std::string, std::function<bool (const std::vector<cmListFileArgument> &, cmExecutionStatus &)>>, std::hash<std::string>, std::equal_to<std::string>>, std::__unordered_map_equal<std::string, std::__hash_value_type<std::string, std::function<bool (const std::vector<cmListFileArgument> &, cmExecutionStatus &)>>, std::equal_to<std::string>, std::hash<std::string>>, std::allocator<std::__hash_value_type<std::string, std::function<bool (const std::vector<cmListFileArgument> &, cmExecutionStatus &)>>>>
     652 objects    87,755,955 bytes  std::unordered_map<std::string, std::function<bool (const std::vector<cmListFileArgument> &, cmExecutionStatus &)>>
  17,239 objects    83,871,867 bytes  std::pair<const std::string, std::function<bool (const std::vector<cmListFileArgument> &, cmExecutionStatus &)>>
 363,885 objects    70,733,073 bytes  std::shared_ptr<const cmStackEntry<const cmListFileContext, cmStackType::Const>>
   8,982 objects    63,414,569 bytes  (anonymous namespace)::cmMacroHelperCommand
 323,312 objects    62,795,168 bytes  cmMakefile::CallScope
 697,026 objects    54,859,728 bytes  std::shared_ptr<const std::string>
   4,104 objects    46,640,909 bytes  std::__hash_table<std::__hash_value_type<cm::String, cmDefinitions::Def>, std::__unordered_map_hasher<cm::String, std::__hash_value_type<cm::String, cmDefinitions::Def>, std::hash<cm::String>, std::equal_to<cm::String>>, std::__unordered_map_equal<cm::String, std::__hash_value_type<cm::String, cmDefinitions::Def>, std::equal_to<cm::String>, std::hash<cm::String>>, std::allocator<std::__hash_value_type<cm::String, cmDefinitions::Def>>>
```

```
// Totals for cmCacheManager::CacheEntry sizeof=80 bytes
// └── 4,282,520 bytes across 82949 allocs
struct cmCacheManager::CacheEntry
{
  std::string                 Value;
  cmStateEnums::CacheEntryType Type;
  cmPropertyMap               Properties; // 4,022,680 bytes across 78,851 allocs : cmPropertyMap
  bool                        Initialized;

  ~cmCacheManager::CacheEntry();
  // directly owned (or unannotated):
  // └── 259,840 bytes across 4,098 allocs
};

// Totals for cmNinjaBuild sizeof=240 bytes
// └── 4,234,664 bytes across 61284 allocs
struct cmNinjaBuild
{
  std::string Comment;
  std::string Rule;
  cmNinjaDeps Outputs; // 331,034 bytes across 9,107 allocs : std::vector<std::string>
  cmNinjaDeps ImplicitOuts;
  cmNinjaDeps WorkDirOuts; // 1,539 bytes across 56 allocs : std::vector<std::string>
  cmNinjaDeps ExplicitDeps; // 317,001 bytes across 6,768 allocs : std::vector<std::string>
  cmNinjaDeps ImplicitDeps; // 45,657 bytes across 270 allocs : std::vector<std::string>
  cmNinjaDeps OrderOnlyDeps; // 131,077 bytes across 3,398 allocs : std::vector<std::string>
  cmNinjaVars Variables; // 3,045,588 bytes across 35,244 allocs : std::map<std::string, std::string>
  std::string RspFile;

  ~cmNinjaBuild();
  // directly owned (or unannotated):
  // └── 362,768 bytes across 6,441 allocs
};

// Totals for std::__tree<std::__value_type<std::string, std::unique_ptr<cmComputeLinkInformation>>, std::__map_value_compare<std::string, std::__value_type<std::string, std::unique_ptr<cmComputeLinkInformation>>, std::less<std::string>>, std::allocator<std::__value_type<std::string, std::unique_ptr<cmComputeLinkInformation>>>> sizeof=24 bytes
```

Caveats

- You need to use clang
- The plugin currently has issues annotating non-inline destructor calls (a fix is in the works)
- If a shared library provides a definition for a destructor (such as `std::string`'s destructor, which is provided by the standard library), this can result in missing annotations.

  Another fix is in the works here.

Final Remarks

- This tool provides increased visibility to memory usage on a per-type and per-object basis
- It makes it a lot more obvious which types are consuming a lot of memory
- It shows a breakdown by *field*, so that you can see which fields are taking up space.

All together, this provides the data necessary to identify inefficiencies, and optimize code for memory usage.

Thank You!