# Heap Dump Format

**Authors:** ruuda@, petrcermak@, primiano@
**Parent doc:** Heap Profiling in Memory-Infra
**Problem:** The heap profiler captures a lot of data, too much to dump into the trace log. How to condense this data into a format that maximises the detail and minimises trace size?
**Approach:** Given what we want to see, what data do we need?



## Recorded heap

Rows represent backtraces, columns represent types.

| | /T | /U | /V | /W | Cum / |
|---|---|---|---|---|---|
| / | 2 | 3 | 5 | 7 | 17 |
| /BrMain | 7 | 11 | 13 | 2 | 33 |
| /BrMain/Init | 151 | 5 | 3 | 83 | 242 |
| /BrMain/MsgLp | 307 | 2 | 281 | 11 | 601 |
| /RdMain | 3 | 7 | 17 | 2 | 29 |
| /RdMain/RTask | 211 | 3 | 5 | 337 | 556 |
| /RdMain/FnA | 2 | 0 | 7 | 11 | 20 |
| /RdMain/FnB | 13 | 3 | 2 | 5 | 23 |
| /ColdFn | 2 | 5 | 7 | 3 | 17 |
| Cum / | 698 | 39 | 340 | 461 | 1538 |
| Cum /BrMain | 465 | 18 | 297 | 96 | 876 |
| Cum /RdMain | 229 | 13 | 31 | 355 | 628 |

Cumulative cells (grey/pink) can be computed. White/yellow cells is what is recorded. Grey and white cells are dumped, yellow/pink cells are not.

# What we want to see

Small values and long tails are not interesting, to save space, merge those into `<other>`. We do want to break down large values. Blue is by backtrace, red is by type.

```
/             1538
  BrMain       876
    Init       242
      T        151
      W         83
      <other>    8
    MsgLp      601
      T        307
      V        281
      <other>   13
    <other>    33
    T          465
      Init     151
      MsgLp    307
      <other>    7
    V          297
      MsgLp    281
      <other>   16
    W           96
      Init      83
      <other>   13
    <other>    18
  RdMain       628
    RTask      556
      T        211
      W        337
      <other>    8
    <other>    72
    T          229
    W          355
    <other>    44
  <other>      34
```

```
/             1538
  T            698
    BrMain     465
      Init     151
      MsgLp    307
      <other>    7
    RdMain     229
      RTask    211
      <other>   18
    <other>     4
  V            340
    BrMain     297
      MsgLp    281
      <other>   16
    <other>    43
  W            461
    BrMain      96
      Init      83
      <other>   13
    RdMain     355
      RTask    337
      <other>   18
    <other>    10
  <other>      39
```

Bad things about this example:

- There never is a big self size (only big sizes are leaf nodes). In general, a backtrace could have `<self>` and `<other>`. In this example, all `<self>` nodes are merged into `<other>`.

Remarks:

- In some cases here, `<other>` consists of only one element. In general, imagine a long tail that is merged into `<other>` as well. Still, if `<other>` consists of one element, it saves space to call it `<other>`, because it need not be dumped. (Parent minus sum of children is size of `<other>`.)

- Sizes displayed are always cumulative sizes; `<self>` is a node, not a different column. If cumulative size is all there is, breaking down is obvious: it breaks down the total into smaller parts that sum to the total. This works if you break down into types, or if you break down into child stack frames.
- This approach works for trees in any dimension. (Types are a tree of height 1.)

# Dump format

Previous format:

- List of “entries”, with a size and optionally a backtrace and/or type.
- Allowed dumping self sizes (values in the white cells), as well as row sums and column sums. Omitting a backtrace or type implies the size is the sum over all backtraces/types.
- The tree structure of backtraces is ignored altogether.
- When omitting values to reduce trace size, unknowns propagate up the tree, so unless everything is dumped, it is difficult to get cumulative sizes.

Proposal:

- Keep the format, but change the meaning of size to mean cumulative size.
- Generate `<self>` nodes before dumping. For example, the backtrace `/BrMain` has size `876`, the backtrace `/BrMain/<self>` has size `33`.
- There is no need to dump `<other>`, its size can be inferred.
- Uniform treatment of different axes (backtrace and type, possibly more in the future).
- Pseudo-format:

```
entries: [
  { size: 1538, bt: [] },
  { size: 876,  bt: [BrMain] },
  { size: 628,  bt: [RdMain] },
  /* <other> backtrace under / is inferred: 1538 - 876 - 628 = 34. */
  { size: 698, bt: [], type: T },
  { size: 340, bt: [], type: V },
  { size: 461: bt: [], type: W },
  /* <other> type for / is inferred: 1538 - 698 - 340 - 461 = 39. */
  { size: 242, bt: [BrMain, Init] },
  { size: 601, bt: [BrMain, MsgLp] },
  /* <other> backtrace under /BrMain is inferred: 876 - 242 - 601 = 33. */
  { size: 151, bt: [BrMain, Init], type: T },
  { size: 83,  bt: [BrMain, Init], type: W },
  /* <other> type for /BrMain/Init is inferred: 242 - 151 - 83 = 8. */
  ...
]
```

- Omitting a type means cumulative size (root node for types). Backtrace will always be specified explicitly.
- See appendix for details.

Backwards compatibility:

- Previous format always starts with `{ size: <total> }` entry, so we can detect the old format.
- For old traces, at import time append `<self>` to every backtrace (because the sizes are self sizes with respect to backtrace). This upgrades the data to the new format.
- If no cumulative size is present in the data, compute it as the sum of the children.
    - Assumes no data is missing, which is the case for old traces.
- Type info was not yet present, no need to deal with that.
- Might remove this after a few releases.

# Appendix: Exact trace log format

See Trace Event Format for context. In the "`dumps`" dictionary that is used for memory dumps, add a "`heaps`" dictionary with a key for every allocator:

```
"dumps" {
    "allocators": { ... },
    "heaps": {
        "partition_alloc": { <see below> },
        "malloc": { <see below> }
    },
    ...
}
```

Every allocator dictionary contains an array "`entries`" (numbers not related to table before):

```
"malloc": {
    "entries": [
        { "size": "511eb9", "bt": "" },
        { "size": "4fcbf1", "bt": "8" },
        { "size": "e398", "bt": "9" },
        { "size": "f06", "bt": "11" },
        { "size": "8c2", "type": "21" },
        { "size": "3a", "bt": "9", "type": "513" },
        ...
    ]
}
```

A entry can have the following properties:

- `size` (required): hexadecimally encoded number of bytes allocated in this context. Includes bytes allocated by more specific contexts.

- `bt` (required): reference to the leaf node of the backtrace in the `stackFrames` dictionary. (See stack traces format in Trace Event Format doc.) For an empty backtrace (root node), this is the empty string.
- `type` (optional): reference to the type name in the `typeNames` dictionary. The `typeNames` dictionary is similar to `stackFrames`, but it does not have a hierarchical structure. When `type` is not specified, size is cumulative over all types. (When types are considered a tree of height 1, this is the root node.)

# Appendix: What to dump

Pseudo algorithm:

1. Compute cumulative sizes of all nodes, across all axes.
   Example: total size is `1538`.
2. For every axis, break down the total size.
   Example:
   - By type: `1538 = 698 (T) + 39 (U) + 340 (V) + 461 (W)`.
   - By backtrace: `1538 = 876 (/BrMain) + 628 (/RdMain) + 17 (/<self>) + 17 (/ColdFn)`.
3. Sort child nodes by size, determine cutoff point. (Or perhaps select children in a smarter way.) Dump cumulative sizes for selected nodes.
   Example: dump types `T`, `V`, `W`, backtraces `/BrMain`, `/RdMain`.
4. Recurse on dumped nodes.
   Example: split `T`, `V`, `W`, by backtrace. Split `/BrMain`, `/RdMain` by backtrace and type.