Using Memory-Backed HDF5 Files to Reduce Storage Access and Size

Gerd Heber

17 December 2020

Contents

1 Introduction

Logically, an HDF5 file stored in a file system is just a sequence of bytes formatted according to the HDF5 file format specification. Several factors affect the performance with which such a sequence can be

manipulated, including (but not limited to) the latency and throughput of the underlying storage hardware. *Memory-backed HDF5 files* eliminate this constraint by maintaining the underlying byte sequence in contiguous RAM buffers.

The life-cycle of memory-backed HDF5 files is very simple:

1. Instruct the HDF5 library to create an HDF5 file in memory rather than in a file system. (It is also possible to load existing HDF5 files into memory buffers.)

2. For the lifetime of the application perform HDF5 I/O operations as usual.

3. Before application shutdown decide whether to abandon the RAM buffer or to persist certain parts of it in an HDF5 file in a file system.

The obvious use case for in-memory HDF5 files is their use as *I/O buffers*, for example, in applications with large numbers of small-size and random-access I/O operations. A second, often overlooked, use case is the ability to deal with a high degree of *uncertainty*, especially, in application backends. The idea is that by maintaining a memory-backed *shadow* HDF5 file for such highly uncertain candidates, the decision about what to put into a persisted (in storage) HDF5 file can often be delayed, resulting in faster access and smaller files.

The vehicle for implementing this behavior in the HDF5 library is the *virtual file layer* (VFL), specifically, the so-called *core* virtual file driver (VFD). This is, however, not the place to learn about implementation details. See ??? for that.

This document is organized as follows: First, we present three examples (sections 2, 3, 4) with their code outlines and their actual behavior. In section 5, we show the implementation of their (common) building blocks.

1.1 Caveat Emptor

The examples in this document are developed by incremental refinement. They are written in C in a style that we call *literate HDF5*. (See Knuth1992.) Each example is first outlined with placeholders inserted for code blocks developed later. That should make the program logic easy to follow, keep the noise-level down, and limit code repetition.

This document (org file) is the *primary* source for different artifacts such as other document formats (HTML, PDF, etc.) *and* the actual source code (`*.c` files). Any change of this document will propagate automatically, making artifacts easy to maintain and to reproduce its results.

This code is for illustration only. For example, no error detection or handling was attempted, and attentive readers will easily identify several corner cases that were not considered.

1.2 Syntax

The basic syntactic element to denote a code block with a deferred implementation follows this pattern:

```
(*
  <<to-be-implemented>> ) (arg1, arg2, arg3, ...);
```

The syntax resembles a C function pointer, where `<<to-be-implemented>>` represents the code implementing the referenced function with arguments `arg1, arg2, arg3`, etc. Readers familiar with *lambda functions* might find it convenient to think about these blocks as such. (If the examples were written in C++, that would be one way to go about it.)

1.3 Logistics

There are several ways to run the examples contained in this document.

1. Emacs users can execute the code blocks containing `main` functions directly via `C-c C-c`, provided the HDF5 library is in their `LD_LIBRARY_PATH` and GCC knows where to find the HDF5 header files and library. If the `h5cc` compiler wrapper is in your `PATH`, execute the following block:

```
(setq org-babel-C-compiler "h5cc --std=gnu99 ")
```

Otherwise, you have to be more specfic. For example:

```
(setq org-babel-C-compiler
      (concat "gcc --std=gnu99 "
              "-I/home/gerdheber/.local/include "
              "-L/home/gerdheber/.local/lib "))
```

2. The examples' source code can be obtained by "tangling" the org file via `C-c C-v t` from Emacs or from the command line by running

```
emacs --batch --eval "(require 'org)" \
      --eval '(org-babel-tangle-file "core-vfd.org")'
```

The code can then be compiled with `gcc --std=gnu99 ...` and the appropriate include and library paths for HDF5.

2 Basic Use

The HDF5 library supports an in-memory, aka. in-*core*, representation of HDF5 files, which should not to be confused with memory-mapped files. Unlike files in a file system, which are represented by inodes and ultimately blocks on a block device, memory-backed HDF5 files are just contiguous memory regions ("buffers") formatted according to the HDF5 file format specification. Since they *are* HDF5 files, the API for such "files" is identical to the one for "regular" (=on-disk) HDF5 files.

The purpose of the first example is to show that working with memory-backed HDF5 files is as straightforward as working with HDF5 files in a file system.

2.1 Goal

We would like to write an array of integers to a 2D dataset in a memory-backed HDF5 file.

Before we look at memory-backed HDF5 files, let's recap the steps for ordinary HDF5 files!

2.2 Outline for an HDF5 file in a file system

Given our important array `data`, we:

1. Create an HDF5 file, `disk.h5`
2. Create a suitably sized and typed dataset `/2x3`, and write `data`
3. Close the dataset
4. Close the file, after printing the HDF5 library version and the file size on-disk

Note: The paragraph following the outline shows the actual program output.

```
 1
 2  <<boilerplate>>
 3
 4  int main(int argc, char** argv)
 5  {
 6    int data[] = {0, 1, 2, 3, 4, 5};
 7    hid_t file = (*
 8                  <<make-disk-file>> ) ("disk.h5");
 9    hid_t dset = (*
10                  <<make-2D-dataset>> ) (file, "2x3", H5T_STD_I32LE,
11                                         (hsize_t[]){2,3}, data);
12    H5Dclose(dset);
13
14    (*
15      <<print-lib-version>> ) ();
16    (*
17      <<print-file-size>> ) (file);
18
19    H5Fclose(file);
20
21    return 0;
22  }
23

HDF5 library version 1.13.0
File size: 4096 bytes
```

The <<boilerplate>> block on line 1 has the usual include directives and is provided in the appendix.

The <<make-disk-file>> block (line 7) is merely a call to H5Fcreate (see section 5.1) and the <<make-2D-dataset>> block (line 9) is a call to H5Dcreate with all the trimmings (see section 5.3).

2.3 Outline for a memory-backed HDF5 file

The outline for memory-backed HDF5 files is almost identical to on-disk files. The <<make-mem-file>> block on line 7 has two additional arguments (see section 5.2). The first is the increment (in bytes) by which the backing memory buffer will grow, should that be necessary. In this example, it's 1 MiB. The third parameter, a flag, controls if the memory-backed file is persisted in storage after closing. Any argument passed to the executable will be interpreted as TRUE and the file persisted. By default (no arguments), there won't be a core.h5 file after running the program.

```
 1
 2  <<boilerplate>>
 3
 4  int main(int argc, char** argv)
 5  {
 6    int data[] = {0, 1, 2, 3, 4, 5};
 7    hid_t file = (*
 8                  <<make-mem-file>> ) ("core.h5", 1024*1024, (argc > 1));
 9    hid_t dset = (*
10                  <<make-2D-dataset>> ) (file, "2x3", H5T_STD_I32LE,
11                                         (hsize_t[]){2,3}, data);
12    H5Dclose(dset);
13
14    (*
15      <<print-lib-version>> ) ();
```

```
16    (*
17      <<print-file-size>> ) (file);
18
19    H5Fclose(file);
20
21    return 0;
22  }
23

HDF5 library version 1.13.0
File size: 1048576 bytes
```

The only difference between the on-disk and the memory-backed version is line 7, which shows that

1. We are dealing with HDF5 files after all.
2. The switch to memory-backed HDF5 files requires only minor changes of existing applications.

See section 5.2 for the implementation of `<<make-mem-files>>`.

2.4 Discussion

When running the executable `core-vfd1` for the memory-backed HDF5 file, we are informed that, for HDF5 library version 1.13.0, the (in-memory) file has a size of 1,048,576 bytes (1 MiB). However, the dataset itself is only about 24 bytes (=six times four bytes plus metadata). Since we told the core VFD to grow the file in 1 MiB increments that's the minimum allocation.

Running the program with any argument will persist the memory-backed HDF5 file as `core.h5`. Surprisingly, that file is only 2072 bytes (for HDF5 1.13.0). The reason is that the HDF5 library truncates and eliminates any unused space in the memory-backed HDF5 file before closing it.

Bottom line: Memory-backed HDF5 files are as easy to use as HDF5 files in file systems.

3 Copying Objects

We can copy HDF5 objects such as groups and datasets inside the same HDF5 file or across HDF5 files. A common scenario is to use a memory-backed HDF5 file as a scratch space (or RAM disk) and, before closing it, to store only a few selected objects of interest in an on-disk HDF5 file.

3.1 Goal

We would like to copy a dataset from a memory-backed HDF5 file to an HDF5 file stored in a file system.

3.2 Outline

In this example, we are working with two HDF5 files, one memory-backed and the other in a file system. We re-use the file creation building blocks (lines 8, 10) and the dataset creation building block (line 12) to create a dataset `dset_m` in the memory-backed HDF5 file `file_m`. Fortunately, the HDF5 library provides a function, `H5Ocopy`, for copying HDF5 objects between HDF5 files. All we have to do is call it on line 21.

```
1
2  <<boilerplate>>
3
4  int main(int argc, char** argv)
5  {
6    int data[] = {0, 1, 2, 3, 4, 5};
7
```

```
 8    hid_t file_d = (*
 9                     <<make-disk-file>> ) ("disk.h5");
10    hid_t file_m = (*
11                     <<make-mem-file>> ) ("core.h5", 4096, (argc > 1));
12    hid_t dset_m = (*
13                     <<make-2D-dataset>> ) (file_m, "2x3", H5T_STD_I32LE,
14                                            (hsize_t[]){2,3}, data);
15    H5Dclose(dset_m);
16
17    (*
18      <<print-lib-version>> ) ();
19    (*
20      <<print-file-size>> ) (file_m);
21
22    H5Ocopy(file_m, "2x3", file_d, "2x3copy", H5P_DEFAULT, H5P_DEFAULT);
23
24    H5Fclose(file_m);
25
26    (*
27      <<print-file-size>> ) (file_d);
28
29    H5Fclose(file_d);
30
31    return 0;
32  }
33

HDF5 library version 1.13.0
File size: 4096 bytes
File size: 4096 bytes
```

3.3 Discussion

When running the program `core-vfd2`, we are informed that, for HDF5 library version 1.13.0, both files have a size of 4 KiB. That is a coincidence of two independent factors: Firstly, in line 10, we instructed the HDF5 library to grow the memory-backed HDF5 file in 4 KiB increments, and one increment is plenty to accommodate our small dataset. Secondly, the 4 KiB size of the `disk.h5` file is due to paged allocation with 4 KiB being the default page size. (*Really?*)

Bottom line: Transferring objects or parts of a hierarchy from a memory-backed HDF5 file to another HDF5 file, be it in a file system or another memory-backed file, is easy thanks to `H5Ocopy`!

4 Delaying Decisions

The developers and maintainers of certain application types, for example, data persistence back-ends of interactive applications, face specific challenges which stem from the *uncertainty* over the particular course of action(s) their users take as part of a transaction or over the duration of a session. Ideally, any decisions that amount to commitments not easily undone later can be postponed or delayed until a better informed decision can be made.

As stated earlier, when creating new objects, the HDF5 library needs certain information (e.g., creation properties) which stays with an object throughout its lifetime and which is immutable. The copy approach from the previous example won't work, because it preserves HDF5 objects' creation properties. Still, a memory-backed HDF5 "shadow" file can be used effectively alongside other HDF5 files as a holding area for objects whose final whereabouts are uncertain at object creation time.

4.1 Goal

We would like to maintain a potentially very large 2D dataset in a memory-backed HDF5 file and eventually persist it to an HDF5 file in a file system.

4.2 Outline

There are a few new snippets in this example. The `<<make-big-2D-dataset>>` block on line 11 appears identical to `<<make-2D-dataset>>`, but the implementation in section 5.5 shows that we are dealing with a datset of potentially arbitrary extent, using chunked storage layout.

Between lines 20 and 24, we mimic the uncertainty around its extent during an application's lifetime by growing and shrinking it using `H5Dset_extent`.

On line 28, we check its size once more (see section 5.6). If the size doesn't exceed 60,000 bytes, we optimize its persisted representation by using the so-called compact storage layout (line 31 and section 5.7). In this case we need to transfer the data manually (line 33 and section 5.8). Otherwise, we fall back onto `H5Ocopy` (line 39).

```
 1
 2  <<boilerplate>>
 3
 4  int main(int argc, char** argv)
 5  {
 6    int data[] = {0, 1, 2, 3, 4, 5};
 7    hid_t file_d = (*
 8                      <<make-disk-file>> ) ("disk.h5");
 9    hid_t file_m = (*
10                      <<make-mem-file>> ) ("core.h5", 1024*1024, (argc > 1));
11    hid_t dset_m = (*
12                      <<make-big-2D-dataset>> ) (file_m, "2x3",
13                                                 H5T_NATIVE_INT32,
14                                                 (hsize_t[]){2,3}, data);
15    (*
16      <<print-lib-version>> ) ();
17    (*
18      <<print-file-size>> ) (file_m);
19
20    { /* UNCERTAINTY */
21      H5Dset_extent(dset_m, (hsize_t[]){200,300});
22
23      H5Dset_extent(dset_m, (hsize_t[]){200000,300000});
24
25      H5Dset_extent(dset_m, (hsize_t[]){2,3});
26    }
27
28    if ((*
29          <<dataset-size>>) (dset_m) < 60000)
30    {
31       hid_t dset_d = (*
32                         <<create-compact>> ) (dset_m, file_d, "2x3copy");
33       (*
34         <<xfer-data>> ) (dset_m, dset_d);
35
36       H5Dclose(dset_d);
37    }
38    else
```

```
39    {
40      H5Ocopy(file_m, "2x3", file_d, "2x3copy", H5P_DEFAULT, H5P_DEFAULT);
41    }
42
43    H5Dclose(dset_m);
44    H5Fclose(file_m);
45
46    (*
47      <<print-file-size>> ) (file_d);
48
49    H5Fclose(file_d);
50
51    return 0;
52  }
53

HDF5 library version 1.13.0
File size: 5242880 bytes
File size: 2048 bytes
```

4.3 Discussion

When running the program `core-vfd3`, we are informed that, for HDF5 library version 1.13.0, the memory-backed HDF5 file has a size of over 4 MiB while the persisted file is just 2 KiB.

As can be seen in section 5.5, the chunk size chosen for the `/2x3` dataset is 4 MiB. Although we are writing only six 32-bit integer (24 bytes), a full 4 MiB chunk needs to be allocated, which explains the overall size for the memory-backed HDF5 file.

The compact storage layout is particularly storage- and access-efficient: the dataset elements are stored as part of the dataset's object header (metadata). This header is read whenever the dataset is opened, and the dataset elements "travel along for free", which means that there is no separate storage access necessary for subsequent read or write operations.

Bottom line: The use of memory-backed HDF5 files can lead to substantial storage and access performance improvements, if applications "keep their cool" and do not prematurely commit storage resources to HDF5 objects.

5 Building Blocks

5.1 On-disk HDF5 file creation

`H5Fcreate` has four parameters, of which the first two, file name and access flag, are usally in the limelight. To create an on-disk HDF5 file is as easy as this:

```
lambda(hid_t, (const char* name),
  {
    return H5Fcreate(name, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
  })
```

The third and the fourth parameter, a *file creation* and a *file access* property list (handle), unlock a few extra treats, as we will see in a moment.

5.2 In-memory HDF5 file creation

We use the fourth parameter of `H5Fcreate`, a file access property list, to do the in-memory magic.

```
 1
 2  lambda(hid_t, (const char* name, size_t increment, hbool_t flg),
 3    {
 4      hid_t retval;
 5      hid_t fapl = H5Pcreate(H5P_FILE_ACCESS);
 6
 7      H5Pset_fapl_core(fapl, increment, flg);
 8
 9      retval = H5Fcreate(name, H5F_ACC_TRUNC, H5P_DEFAULT, fapl);
10      H5Pclose(fapl);
11      return retval;
12    })
13
```

That's right, a suitably initialized property list (line 6) makes all the difference. This is in fact the ONLY difference between an application using regular vs. memory-backed HDF5 files.

5.3 Dataset creation

To create a dataset, we must specify a `name`, its element type `dtype`, its shape `dims`, and, optionally, an inital value `buffer`. Without additional customization, the default dataset storage layout is `H5D_CONTIGUOUS`, i.e., the (fixed-size) dataset elements are layed out in a contigous (memory or storage) region.

```
 1
 2  lambda(hid_t,
 3         (hid_t file, const char* name, hid_t dtype, const hsize_t* dims, void* buffer),
 4    {
 5      hid_t retval;
 6      hid_t dspace = H5Screate_simple(2, dims, NULL);
 7
 8      retval = H5Dcreate(file, name, dtype, dspace,
 9                         H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
10
11      if (buffer)
12        H5Dwrite(retval, dtype, H5S_ALL, H5S_ALL, H5P_DEFAULT, buffer);
13
14      H5Sclose(dspace);
15      return retval;
16    })
17
```

WARNING: This snippet contains an *important assumption* that may not be obvious to many readers: The datatype handle `dtype` is used in two places with different interpretations. In the first instance, line 7, it refers to the in-file element type of the dataset to be created. In the second instance, line 11, it refers to the datatype of the elements in `buffer`. The assumption is that the two are the same. While this assumption is valid in many practical examples, it can lead to subtle errors if its violation goes undetected. In a production code, this should be either documented and enforced, or an additional datatype argument be passed to distinguish them.

5.4 Print library and file info

```
lambda(void, (void),
  {
    unsigned majnum;
    unsigned minnum;
    unsigned relnum;
    H5get_libversion(&majnum, &minnum, &relnum);
    printf("HDF5 library version %d.%d.%d\n", majnum, minnum, relnum);
  })
```

```
lambda(void, (hid_t file),
  {
    hsize_t size;
    H5Fget_filesize(file, &size);
    printf("File size: %ld bytes\n", size);
  })
```

5.5 Big dataset creation

This `lambda` returns a handle to the potentially large dataset in the memory-backed HDF5 file. Since the dataset's final size will only be known eventually (e.g., end of epoch or transaction), we can't impose a finite maximum extent. On line 6, we set the maxmimum extent as unlimited in all (2) dimensions. Currently, the only HDF5 storage layout that supports such an arrangement is *chunked storage layout*. By passing a non-default dataset creation property list `dcpl` to `H5Dcreate` (line 11), we instruct the HDF5 library to use chunked storage layout instead of the default contiguous layout. For chunked layout, we must specify the size of an individual chunk in terms of *dataset elements per chunk*; see line 9. The size of a chunk in bytes depends on the element datatype. In our example (32-bit integers), a 1024^2 chunk occupies 4 MiB of memory or storage.

```
 1
 2  lambda(hid_t,
 3    (hid_t file, const char* name, hid_t dtype, const hsize_t* dims, void* buffer),
 4    {
 5      hid_t retval;
 6      hid_t dspace = H5Screate_simple(2, dims,
 7                                      (hsize_t[]){H5S_UNLIMITED, H5S_UNLIMITED});
 8      hid_t dcpl = H5Pcreate(H5P_DATASET_CREATE);
 9
10      H5Pset_chunk(dcpl, 2, (hsize_t[]){1024, 1024});
11      retval = H5Dcreate(file, name, dtype, dspace,
12                         H5P_DEFAULT, dcpl, H5P_DEFAULT);
13
14      if (buffer)
15        H5Dwrite(retval, dtype, H5S_ALL, H5S_ALL, H5P_DEFAULT, buffer);
16
17      H5Pclose(dcpl);
18      H5Sclose(dspace);
19      return retval;
20    })
21
```

The same warning and assumptions expressed at the end of section 5.3 apply to `dtype`.

5.6 Dataset size

This `lambda` returns the size (in bytes) of the source dataset in the memory-backed HDF5 file. It's a matter of determining the storage size of an individual dataset element and counting how many there are (lines 7, 8)

```
 1
 2  lambda(hid_t, (hid_t dset),
 3    {
 4      size_t retval;
 5      hid_t ftype = H5Dget_type(dset);
 6      hid_t dspace = H5Dget_space(dset);
 7
 8      retval = H5Tget_size(ftype) *
 9                 (size_t) H5Sget_simple_extent_npoints(dspace);
10
11      H5Sclose(dspace);
12      H5Tclose(ftype);
13      return retval;
14    })
15
```

5.7 Compact replica

This `lambda` returns a handle to the freshly minted compact replica of the source dataset. (It's a placeholder, because the actual values are transferred separately.)

What sets this dataset creation apart from the default case occurs on lines 14-16. By passing a non-default dataset creation property list `dcpl` to `H5Dcreate`, we instruct the HDF5 library to use compact storage layout instead of the default contiguous (`H5D_CONTIGUOUS`) layout.

```
 1
 2  lambda(hid_t, (hid_t src_dset, hid_t file, const char* name),
 3    {
 4      hid_t retval;
 5      hid_t ftype = H5Dget_type(src_dset);
 6      hid_t src_dspace = H5Dget_space(src_dset);
 7      hid_t dcpl = H5Pcreate(H5P_DATASET_CREATE);
 8
 9      hid_t dspace = H5Scopy(src_dspace);
10      hsize_t dims[H5S_MAX_RANK];
11      H5Sget_simple_extent_dims(dspace, dims, NULL);
12      H5Sset_extent_simple(dspace, H5Sget_simple_extent_ndims(dspace),
13                           dims, NULL);
14
15      H5Pset_layout(dcpl, H5D_COMPACT);
16      retval = H5Dcreate(file, name, ftype, dspace,
17                         H5P_DEFAULT, dcpl, H5P_DEFAULT);
18
19      H5Pclose(dcpl);
20      H5Sclose(dspace);
21      H5Tclose(ftype);
22      return retval;
23    })
24
```

Two other things are worth mentioning about this snippet.

1. The dataspace construction on lines 8-12 appears a little clumsy. Since the extent of the source dataset `src_dset` is not changing, why not just work with `src_dspace` (line 5)? The reason is that dataspaces with `H5S_UNLIMITED` extent bounds, for obvious reasons, are not supported with compact layout. In that case, in our example (!), passing `src_dspace` as an argument to `H5Dcreate` would generate an error. It's easier to just create a copy of the dataspace and "kill" (`NULL`) whatever maximum extent there might be.

2. On line 9, we use the HDF5 library macro `H5S_MAX_RANK` to avoid the dynamic allocation of the `dims` array.

5.8 Data transfer

The HDF5 library does not currently have a function to "automagically" transfer data between two datasets, especially datasets with different storage layouts. There is not much else we can do but to read (line 8) the data from the source dataset and to write (line 9) to the destination dataset.

Since we transfer the data through memory, we need to determine first the size of the transfer buffer needed (line 5).

```
 1
 2  lambda(void, (hid_t src, hid_t dst),
 3    {
 4      hid_t ftype = H5Dget_type(src);
 5      hid_t dspace = H5Dget_space(src);
 6      size_t size = H5Tget_size(ftype) * H5Sget_simple_extent_npoints(dspace);
 7      char* buffer = (char*) malloc(size);
 8
 9      H5Dread(src, ftype, H5S_ALL, H5S_ALL, H5P_DEFAULT, buffer);
10      H5Dwrite(dst, ftype, H5S_ALL, H5S_ALL, H5P_DEFAULT, buffer);
11
12      free(buffer);
13      H5Sclose(dspace);
14      H5Tclose(ftype);
15    })
16
```

6 Appendix

6.1 Versions

This document was tested with the following software versions:

```
(princ (concat
        (format "Emacs version: %s\n"
                (emacs-version))
        (format "org version: %s\n"
                (org-version))))
```

```
Emacs version: GNU Emacs 26.1 (build 2, x86_64-pc-linux-gnu, GTK+ Version 3.24.5)
 of 2019-09-22, modified by Debian
org version: 9.1.9
```

```
gcc --version
```

```
gcc (Debian 8.3.0-6) 8.3.0
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

6.2 Boilerplate with a twist

These are the header files needed to build the examples.

```
#include "hdf5.h"

#include <stdio.h>
#include <stdlib.h>
```

The more interesting bit is the `lambda` macro by Al Williams.

```
1
2  #define lambda(lambda$_ret, lambda$_args, lambda$_body) \
3    ({                                                    \
4      lambda$_ret lambda$__anon$ lambda$_args             \
5        lambda$_body                                      \
6        &lambda$__anon$;                                  \
7    })
8
```

It uses two features of GNU C (`--std=gnu99`), namely, nested functions and statement expressions, which lets us wrap C code blocks as "lambda functions", thereby making longer pieces of code easier to follow and digest.

```
lambda(<return type>, ([type1 arg1, type2 arg2, ...]), { <lambda body>  })
```

Such a `lambda` can then be invoked like a C-function pointer:

```
 1
 2  #define lambda(lambda$_ret, lambda$_args, lambda$_body) \
 3    ({                                                    \
 4      lambda$_ret lambda$__anon$ lambda$_args             \
 5        lambda$_body                                      \
 6        &lambda$__anon$;                                  \
 7    })
 8
 9  int main()
10  {
11    printf("%f\n",(*lambda(float, (float x), { return x*x; }))(2.0));
12    return 0;
13  }
14

4.0
```