The LEDA memory manager

Sven Thiel

August 19, 1998

Abstract

The following paper describes the LEDA memory manager which is used to speed up allocating and freeing memory. It is much faster than the built-in `new` and `delete` routines, especially when many small chunks of memory are allocated.

1 Overview

The main idea to achieve the speed up is the following: When many comparatively small chunks of memory are required by an application the memory manager does not call `malloc` for each chunk but it calls `malloc` only once to allocate a huge block of memory; this block is sliced into small chunks of equal size which are distributed to the application. Only when there are no free chunks left, the manager has to allocate a new block of memory. Thus the expensive call of the library function is amortized. When the application does not need a chunk anymore and returns it back to the memory manager, it is placed into the list of free chunks. The manager does not return any memory to the operating system by a call to `free` until the end of the application. This does not only avoid expensive calls of library functions but it also prevents the heap from becoming fragmented.
The LEDA memory management can be divided into three parts:

1. Code which provides a standard manager and ensures that all translation units, i.e. object files, of an application share a single instance of the memory manager class as standard manager.
2. Code that realizes the actual memory management as outlined above.
3. The macro `LEDA_MEMORY(...)` which provides an easy way to make user-defined classes use the LEDA memory management.

2 Details

2.1 Providing the standard manager

As mentioned above there is only one standard memory manager which is called *std_memory_mgr*, even though there can be more than one translation unit using that memory manager. Therefore the manager must be a single global object which is defined in one translation unit and referenced in all other units which use it. Of course, the memory manager must be initialized before its first use. Unfortunately, C++ does neither guarantee that a global object is constructed before its first use nor that is destructed after its last use. Because of this, LEDA uses a trick which some C++-implementations use to make sure that there is only one instance of the streams *cout*, *cin* and *cerr* and that the streams are fully constructed before their first use. The main idea is to use an auxiliary class serving as a counter for the number of translation units which are referencing the memory manager. (A translation unit can reference the manager by including the LEDA header file `memory.h`.)

Then the manager can be constructed when the counter is incremented for the first time and destructed when the counter reaches zero again.

⟨*memory.h*⟩≡

```
class __exportC memory_manager_init {

public:

static unsigned count;

  memory_manager_init();
 ~memory_manager_init();
};
```

In order to make the counter work every translation unit must define its own instance of *memory_manager_init*. It is declared `static` to make it local to the respective translation unit so that there can be no name conflict between several units.

⟨*memory.h*⟩+≡

```
static memory_manager_init memory_mgr_init;
```

Finally, here comes the code which defines the standard memory manager and effectively constructs and destructs it as described above. (As `count` has no explicit initializer, it will be initialized to zero before its first use.)

⟨*_memory.c*⟩≡

```
memory_manager std_memory_mgr;

unsigned memory_manager_init::count;

memory_manager_init::memory_manager_init()
{ if (count++ == 0)
  {
    ...
    std_memory_mgr.init(STDMEM_TABLESIZE,"STD_MEMORY_MGR");
    ...
  }
}

memory_manager_init::~memory_manager_init()
{ if (--count == 0) std_memory_mgr.destroy(); }
```

As one can see below, the constructor and the destructor of the *memory_manager* class do nothing if the standard manager is constructed or destructed and else they call the function `init` or `destroy` to perform the work to be done. These functions will be described later.

⟨*_memory.c*⟩+≡

```
memory_manager::memory_manager(int sz)
{ if (this != &std_memory_mgr) init(sz,"USR_MEMORY_MGR"); }

memory_manager::~memory_manager()
{ if (this != &std_memory_mgr) destroy(); }
```

2.2 The actual memory management

The data structures

The strategy of allocating large blocks and slicing them into chunks is only efficient if the size of the chunks is small compared to size of the block. Therefore the memory manager does not apply this strategy in case the application requests a large chunk of memory; such a large chunk is called a *vector*. The memory manager records requests for allocation and deallocation of vectors in its statistics, but these requests are directly mapped to calls to `malloc` and `free`. The macro `STDMEM_TABLESIZE` (currently set to 256) defines the lower bound on the size (in bytes) of vectors for the standard memory manager.
In order to be able to apply the slicing technique, the memory manager maintains two tables.

- The first table is the **block-list table**, it is used to keep track of all allocated memory blocks. A sketch of this table can be seen in figure 1.

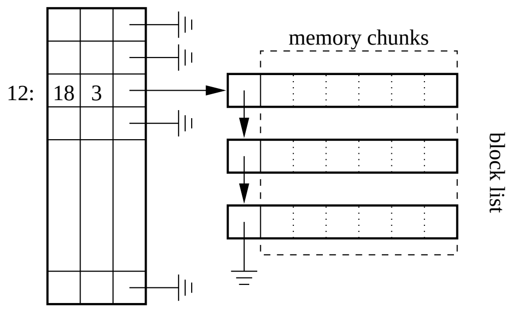

Figure 1: block-list table
The example shows 18 chunks of size 12 allocated in 3 blocks.

  For every possible size i of a small chunk the table contains three entries:

  1. `total_count`: the total number of chunks (usable for the application)
  2. `block_count`: the total number of blocks
  3. `block_list`: pointer to the head of the block-list
     The first word of the first chunk of every block is used to link the blocks to a list. These chunks are not distributed to the application.

  The first two entries in line 0 of the table are used to keep statistics of the vectors: the entry `total_count` stores the number of currently allocated bytes for vectors, the entry `block_count` contains the total number of vectors which have ever been allocated. This table is used for maintaining statistics and for cleaning up at the end of the application.

- The second table, the **free-list table**, is used for distributing memory chunks to the application and to collect unused chunks from it; a sketch of this table can be found in figure 2.

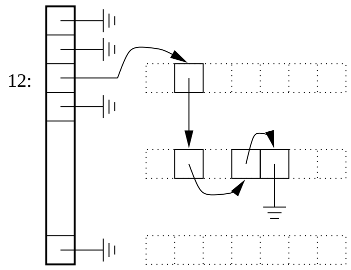

Figure 2: free-list table (continues the example from above)
Only 4 chunks are in the free-list, the others are used by the application.

The ith entry in the table contains a pointer to the first chunk in the list of all free chunks of size i. The first word of every free chunk is used as a link to the next free chunk of the same size. When a free chunk is given to the application the link is not needed any more and the memory can be overwritten. Thus the free-lists do not cost any space overhead (except for the space needed by the second table).

Allocating and deallocating small chunks

When the memory manager has to process a request for a chunk of size i, it first checks whether the ith entry of the free-list table is *nil*. If it is, the manager has to allocate a new block of memory and slice it into chunks. The size of the block is independent of i, it is a constant named `block_bytes`. This constant is chosen as a power of two minus one word because many implementations of `malloc` increase the size of the allocated block by one word to store bookkeeping information and then round it to the next power of two. As mentioned before, the block returned by `malloc` is then sliced into equally sized chunks. The size of these chunks is not exactly i but the smallest multiple of the word size which is not less than i. This is done because many systems can access memory much faster if it its beginning coincides with the beginning of a machine word (word aligning). The first chunk is used to link the blocks together in the block list, the remaining chunks are inserted in the free-list. Now, the free-list for i contains at least one chunk. The first chunk is popped from the list and given to the application.

When the application returns a chunk of size i to manager, the chunk is simply made the head of the free-list and the former head becomes its successor.

The main member functions of the *memory_manager* class

- `init(int sz, const char* name = 0)`:
  This function performs the actual construction of the manager. The argument `sz` specifies the size of the two tables, i.e. the lower bound on the size of chunks that are considered as vectors.
- `destroy()`:
  This function carries out the actual destruction. It checks the option list of LEDA for the flags *show_memory_leaks* and *show_memory_statistics* and calls the function `print_used_memory` or `print_statistics` if appropriate. (The option list is read from the environment variable *LEDA_INIT*.) After that it calls `clear` and finally it frees the memory occupied by the two tables.
- `allocate_block(size_t s, int N)`:
  This function allocates a block of size `block_bytes` (if $N < 0$) or of size $s \cdot (N + 1)$, slices it into chunks of size s and adds the block to the block-list and the new chunks to the free-list.
  Precondition : $1 \leq s < max\_sz$
- `allocate_bytes(size_t s) / deallocate_bytes(void *, size_t s)`:
  The function (de)allocates a chunk of size s; if s is less than max_sz, then the allocation strategy from above is applied, otherwise the work is handed down to `(de)allocate_vector`.
- `allocate_bytes_with_check(...) / deallocate_bytes_with_check(...)`:
  These are the debugging variants of the functions above. On deallocation it is checked for example if the given pointer is deleted twice.
- `allocate_vector(size_t s) / deallocate_vector(void*)`:
  The function is used to (de)allocate large chunks; apart from some statistical book-keeping, it simply performs a call to `malloc` / `free`.
- `clear()`:
  For every i in $[1; sz[$ this function checks whether all chunks of size i are contained in the respective free-list. If this is the case, then all blocks in the ith block-list are freed. Thus this function can be used safely at runtime to force the memory manager to shrink its memory usage.

- `kill()`:
  This function frees all blocks in all block-lists no matter if some of the memory is still occupied by the application or not. Therefore this function should normally not be used before the application terminates.

- `print_statistics() / print_used_memory()`:
  Both functions have in common that they print statistical information. The first function prints information about the chunks of size i whenever the ith block-list contains at least one block and it prints information about vectors. The second function only prints information about small chunks; it outputs information about the chunks of size i only if at least one of these chunks is still used by the application. In case that there are no chunks occupied by the application, i.e. all chunks are in the free list, the function does not print anything at all. Therefore it is predestined to check for memory leaks when the application terminates. It gives more accurate results if it is not called explicitly but triggered by the flag *show_memory_leaks* in the LEDA option list (see also the description of `destroy` on page 4), for then it is called after the clean-up of the static data members of certain classes has been done.

  The output of both functions is a table which consists of 5 columns:

  1. size: This is the size of the chunks for which the following information is reported.
  2. used: The number of chunks which are currently used by the application.
  3. free: The number of chunks which are currently in the free-list, i.e. the number of chunks that have been allocated by the memory manager but that are not used by the application.
  4. blocks: The number of blocks that are currently in the block-list.
  5. bytes: $size \cdot (used + free)$

2.3 The macros `LEDA_MEMORY(...)` and `LEDA_MEMORY_WITH_CHECK(...)`

These macros define the operators `new` and `delete` for the class to which they are applied. The latter macro simply maps these operators to a call to the manager function `(de)allocate_bytes_with_check`. The first macro, however, does not call `(de)allocate_bytes` as one might expect, but the code which it defines for `new` and `delete` is nearly the same as the code for the two functions. The reason for this hand-made inlining is that the decision whether a chunk is to be considered as a small chunk or as a vector can be made at compile time.