Описание библиотеки stack_heap

17 декабря 2012 г.

Содержание

1 Назначение библиотеки **2**

2 Основные достоинства библиотеки **2**

3 Недостатки библиотеки **2**

4 API библиотеки **3**

5 Пример использования **5**
5.1 Инициализация, использование, завершение 5
5.2 Поиск неосвобождённой памяти 6
5.3 Создание вложенных куч с использованием stack_heap . . . 7

6 Описание архитектуры библиотеки **8**

7 Лист изменений документа **11**

1 Назначение библиотеки

Библиотека stack_heap предназначена для замены стандартных процедур работы с динамической памятью. Основное отличие этой библиотеки от стандартных средств заключается в том, что библиотека даёт больше возможностей как по контролю за использованием динамической памяти, так и по решению возможных проблем с её утечками. Библиотека призвана дать примерно те же возможности по отладке, что даёт библиотека dmalloc, но при этом она позволяет решить проблему возврата потерянной памяти не особо разбираясь, где эта память должна быть освобождена.

2 Основные достоинства библиотеки

- Поддержка нескольких куч (это особенно актуально для систем, где есть разные виды памяти).
- Поддержка вложенных куч - это даёт возможность использовать отдельную кучу для какой-нибудь задачи, а по окончании выполнения задачи уничтожить кучу, не заботясь об утечках памяти.
- Поддержка возможности сбора статистики по куче (список неосвобождённых указателей, объём общей памяти, объём неосвобождённой памяти).
- Защита от ошибок программиста. Библиотека постоянно анализирует целостность кучи и в случае обнаружения проблем выдаёт сообщение об ошибке. Также она защищает программиста от повторных освобождений одного и того же указателя.
- Поддержка произвольного выравнивания. Библиотека позволяет выделять блоки памяти с любым выравниванием (в т.ч. не степенями 2).
- Портабельность. Библиотека в настоящее время собирается для ОС Linux и для DSP-проектов.

3 Недостатки библиотеки

- Библиотека пока что не использует каких-либо специфических структур данных типа binary heap для ускорения поиска подходящего элемента. Т.е. аллокация памяти может занимать время O(n) где n - количество свободных блоков. Таким образом, она плохо подойдёт для применений, где выделяется много маленьких кусочков памяти и где при этом, критично время выделения этой памяти.
- Одна структура данных (заголовок блока), соответствующая блоку, занимает 24 байта (6 слов) в случае, если отключены отладочные возможности и 36 байт (9 слов) в случае, если отладочные возможности включены. Это также делает применение библиотеки малоцелесообразным, если размеры самих выделяемых кусочков сравнимы с размерами заголовков блоков.

4 API библиотеки

```
sth_t * sth_init(
        void * address,
        unsigned long size,
        sth_dbg_t *dbg);
```

Создаёт новую кучу. Идентификатором кучи будет переменная-указатель на sth_ctx_t. Входные параметры:
address - указатель на начало области памяти, отводимой под кучу
size - размер области памяти, отводимой под кучу
dbg - указатель на отладочную структуру для этой кучи (можно оставлять равным NULL).

```
int         sth_set_name(
        sth_t * heap,
        const char * const name);
```

Присваивает куче имя. Входные параметры:
heap - указатель на структуру кучи, для которой задаётся имя
name - указатель на имя кучи

Куче можно присвоить имя. Имеет смысл сделать это, если куч несколько и хочется, чтоб отладочная информация содержала имя кучи. Эта функция будет работать только если куча связана с отладочной структурой.

```
sth_dbg_t *sth_dbg_init(
        sth_t * heap,
        int hash_table_size);
```

Инициализирует отладочную структуру. Входные параметры:
heap - указатель на структуру кучи, которая будет использоваться для работы отладочной структуры
hash_table_size - размер хэш-таблицы

Отладочная структура при своей работе требует выделения памяти, поэтому ей нужно предоставить указатель на кучу, с которой она будет работать. При этом эта отладочная структура становится отладочной структурой для этой кучи, т.е. допустима такая последовательность создания объектов:

```
sth_t *heap = sth_init(some_address, some_size, NULL);
sth_dbg_t *dbg = sth_dbg_init(heap, 150);
```

при этом отладочная функциональность, контекстом которой является структура dbg будет использовать heap в качестве динамической памяти, а сама heap (несмотря на то, что у неё в качестве указателя на отладочную структуру был указан NULL, будет использовать dbg в качестве своей отладочной структуры)

```
void sth_pf_init(printf_t pf);
```

Входные параметры:
pf - указатель на функцию, которая имеет прототип в точности соответствующий функции printf

Такую функцию необходимо предоставить библиотеке, чтоб она могла сообщать об ошибках и выводить другую информацию.

```
void      * sth_malloc(
          sth_t * heap,
          unsigned long size,
          int align,
          const char * const file_name,
          const char * const fn_name,
          const unsigned line_idx);
```

Эквивалент malloc. Не очень удобен, т.к. принимает много параметров, половина из которых не относится к аллокации памяти и служит только для отладки. Поэтому добавлен макрос:

```
#define STH_MALLOC(HEAP, SIZE, ALIGN)       \
          sth_malloc((HEAP),(SIZE),(ALIGN), __FILE__, __FUNCTION__, __LINE__)
```

```
void        sth_free(
          void * ptr,
          const char *const file_name,
          const char *const fn_name,
          const unsigned line_idx);
```

Эквивалент free. Не очень удобен по той же причине, что и sth_malloc. Поэтому добавлен макрос:

```
#define STH_FREE(PTR) \
          sth_free((PTR), __FILE__, __FUNCTION__, __LINE__)
```

```
void        sth_print_stat(sth_t * heap);
```

Выводит статистику об использовании кучи, о количестве блоков, количестве общей памяти, количестве аллоцированной памяти. Для вывода использует функцию, переданную в библиотеку через sth_pf_init.

```
sth_stat_t sth_get_stat(sth_t * heap);
```

возвращает статистику об использовании кучи в виде структуры.

```
void        sth_print_heap_structure(sth_t * heap);
```

выводит информацию о структуре кучи (содержимое полей заголовка и блоков памяти кучи).

5 Пример использования

5.1 Инициализация, использование, завершение

```
void init_heaps(gl_dsp_config_t *cfg)
{
    SYS_heap_stat_t hs;
    const int DS_RAM_RESERVED = 100000;
    const int INTERN_RESERVED =   2500;
    const int PM_RESERVED     =   2500;
    const int S_RESERVED      =   2500;

    SYS_HeapStat(&hs, SYS_MemDSRAM);
    int ds_ram_alloc_size = hs.heap_max_cont_free - DS_RAM_RESERVED;
    SYS_HeapStat(&hs, SYS_MemIntern);
    int intern_alloc_size = hs.heap_max_cont_free - INTERN_RESERVED;
    SYS_HeapStat(&hs, SYS_MemPM_RAM);
    int pm_alloc_size = hs.heap_max_cont_free - PM_RESERVED;
    SYS_HeapStat(&hs, SYS_MemSRAM);
    int s_alloc_size = hs.heap_max_cont_free - S_RESERVED;

    void *ptr_dsram  = SYS_malloc(ds_ram_alloc_size, SYS_MemDSRAM);
    void *ptr_intern = SYS_malloc(intern_alloc_size, SYS_MemIntern);
    void *ptr_pm     = SYS_malloc(pm_alloc_size,     SYS_MemPM_RAM);
    void *ptr_s      = SYS_malloc(s_alloc_size,      SYS_MemSRAM);

    sth_pf_init(PrintDbg);

    cfg->heap_ds_ram =
        sth_init(ptr_dsram, ds_ram_alloc_size, NULL);
    cfg->stack_heap_dbg =
        sth_dbg_init(cfg->heap_ds_ram, 129);
    cfg->heap_intern =
        sth_init(ptr_intern, intern_alloc_size, cfg->stack_heap_dbg);
    cfg->heap_pm_ram =
        sth_init(ptr_pm, pm_alloc_size, cfg->stack_heap_dbg);
    cfg->heap_sram =
        sth_init(ptr_s, s_alloc_size, cfg->stack_heap_dbg);
}

void done_heaps(gl_dsp_config_t *cfg)
{
    SYS_free(cfg->heap_sram);
    SYS_free(cfg->heap_pm_ram);
    SYS_free(cfg->heap_intern);
    SYS_free(cfg->heap_ds_ram);
}

void some_function(gl_dsp_config_t *cfg)
{
    const int MEMORY_SIZE = 10000;
    int *memory = STH_MALLOC(cfg->heap_intern, MEMORY_SIZE, 1);
    ..................
    STH_FREE(cfg->heap_intern);
}
```

Константы DS_RAM_RESERVED и подобные нужны для того, чтоб передать библиотеке не всю память (чтоб GidrOS могла ещё что-то аллоцировать с помощью обычного SYS_malloc).

5.2 Поиск неосвобождённой памяти

```
void init_heap(gl_dsp_config_t *cfg)
{
    SYS_heap_stat_t hs;
    const int DS_RAM_RESERVED = 100000;

    SYS_HeapStat(&hs, SYS_MemDSRAM);
    int ds_ram_alloc_size = hs.heap_max_cont_free - DS_RAM_RESERVED;

    void *ptr_dsram  = SYS_malloc(ds_ram_alloc_size, SYS_MemDSRAM);

    sth_pf_init(PrintDbg);

    cfg->heap_ds_ram =
        sth_init(ptr_dsram, ds_ram_alloc_size, NULL);
    cfg->stack_heap_dbg =
        sth_dbg_init(cfg->heap_ds_ram, 129);
}

void done_heap(gl_dsp_config_t *cfg)
{
    SYS_free(cfg->heap_ds_ram);
}

void some_function(gl_dsp_config_t *cfg)
{
    const int MEMORY_SIZE = 10000;
    int *memory = STH_MALLOC(cfg->heap_ds_ram, MEMORY_SIZE, 1);
    ..................
    STH_FREE(cfg->heap_ds_ram);
    sth_print_stat(cfg->heap_ds_ram);
}
```

sth_print_stat выводит в отладочный поток данных информацию о куче, включая список неосвобожденных блоков с указанием мест, где они были аллоцированы.

5.3 Создание вложенных куч с использованием stack_heap

```
void init_heap(gl_dsp_config_t *cfg)
{
    SYS_heap_stat_t hs;
    const int DS_RAM_RESERVED = 100000;

    SYS_HeapStat(&hs, SYS_MemDSRAM);
    int ds_ram_alloc_size = hs.heap_max_cont_free - DS_RAM_RESERVED;

    void *ptr_dsram  = SYS_malloc(ds_ram_alloc_size, SYS_MemDSRAM);

    sth_pf_init(PrintDbg);

    cfg->heap_ds_ram =
        sth_init(ptr_dsram, ds_ram_alloc_size, NULL);
    cfg->stack_heap_dbg =
        sth_dbg_init(cfg->heap_ds_ram, 129);
}

void done_heap(gl_dsp_config_t *cfg)
{
    SYS_free(cfg->heap_ds_ram);
}

void some_other_function(sth_t * heap)
{
    const int MEMORY_SIZE = 100;
    int *memory = STH_MALLOC(heap, MEMORY_SIZE, 1);
    .........
    int *memory = STH_MALLOC(heap, MEMORY_SIZE, 1);
    .........
    int *memory = STH_MALLOC(heap, MEMORY_SIZE, 1);
}

void some_function(gl_dsp_config_t *cfg)
{
    const int NESTED_HEAP_SIZE = 10000;
    sth_t *nested_heap = STH_MALLOC(cfg->heap_ds_ram, NESTED_HEAP_SIZE, 1);

    some_other_function(nested_heap);

    STH_FREE(nested_heap);
}
```

В данном примере some_other_function намеренно написана некорректно, в ней происходит утечка памяти, но при этом, после возврата в some_function вся потерянная память возвращается.

В реальности так писать программы нельзя, однако такая функциональность кучи может быть полезна, т.к. иногда (например, при хранении строчек, созданных командой аналогичной strdup) хранить все указатели с целью потом освободить память может быть неоправданно.

6 Описание архитектуры библиотеки

Библиотека состоит из двух модулей: stack_heap.c и str_hash.c В них размещены следующие классы (см рис.1):

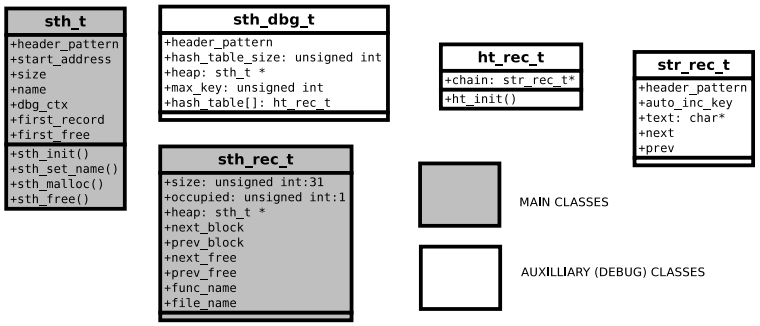

Рис. 1: Диаграмма классов

После вызова sth_init, память кучи выглядит как показано на рис.2

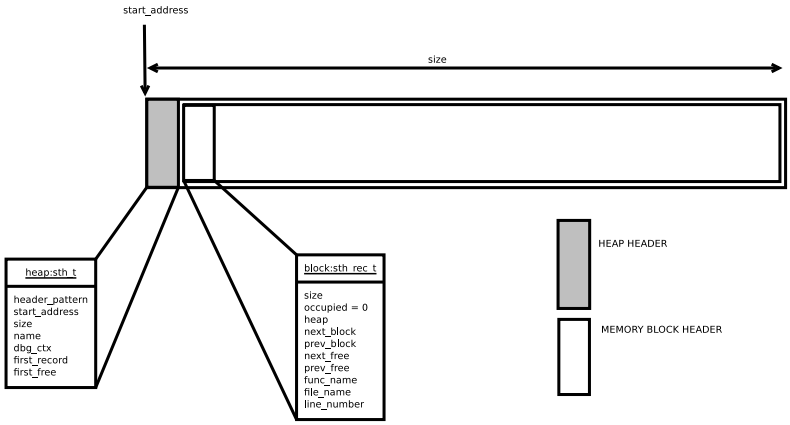

Рис. 2: Использование памяти менеджером кучи сразу после инициализации

После выделения первого блока память выглядит как показано на рис.3

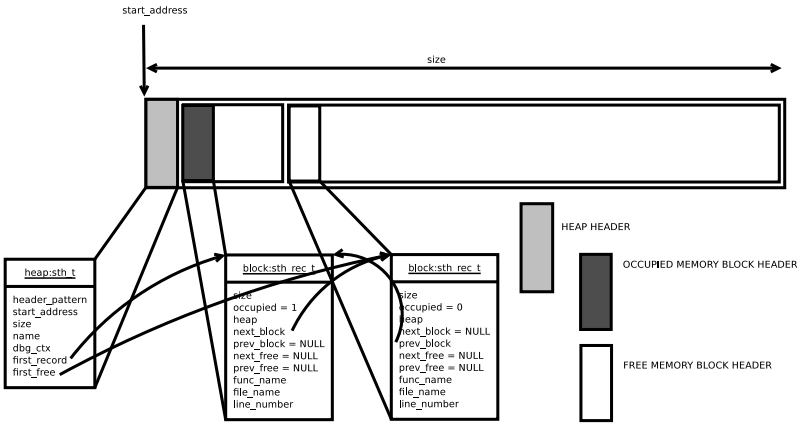

Рис. 3: Использование памяти менеджером кучи после выделения первого блока

В случае, если отладочные функции не включены (например, библиотека скомпилирована с опцией STH_NO_DEBUG_INFORMATION), то будут искользоваться только классы sth_t и sth_rec_t.

В случае, если отладочные функции включены, то для выделения памяти под экземпляры дополнительных классов будет использоваться функция sth_malloc.

Смысл дополнительных классов в том, чтоб организовать такую структуру данных, чтоб каждую строчку хранить только один раз (т.е. например, если будет необходимо 3 раза сохранить в памяти строчку «complex_signal.c» то при первом вызове функции

```
const char * ht_reg_string(
    const char * const       text,
    struct sth_dbg_tag * dbg);
```

под эту строчку действительно будет выделена память и будет возвращён указатель на выделенную строчку, но при последующих вызовах выделяться память не будет, а будет возвращаться указатель на существующую строчку).

Такое поведение достигается за счёт использования хэш-таблицы: для каждой регистрируемой строчки считается значение хэш-функции (для этого перемножаются все символы строчки кроме завершающего нулевого и делятся с остатком на размер хэш-таблицы (задаётся в функции sth_dbg_init). Соответствующая строчка хэш-таблицы содержит указатель на список, элементами которого являются строки с одинаковым значением хэша. Т.к. вероятность одинакового значения хэша (при разном содержимом) мала, то поиск внутри списка занимает мало времени (поиск же в хэш-таблице занимает только время, необходимое для расчёта хэш-функции).

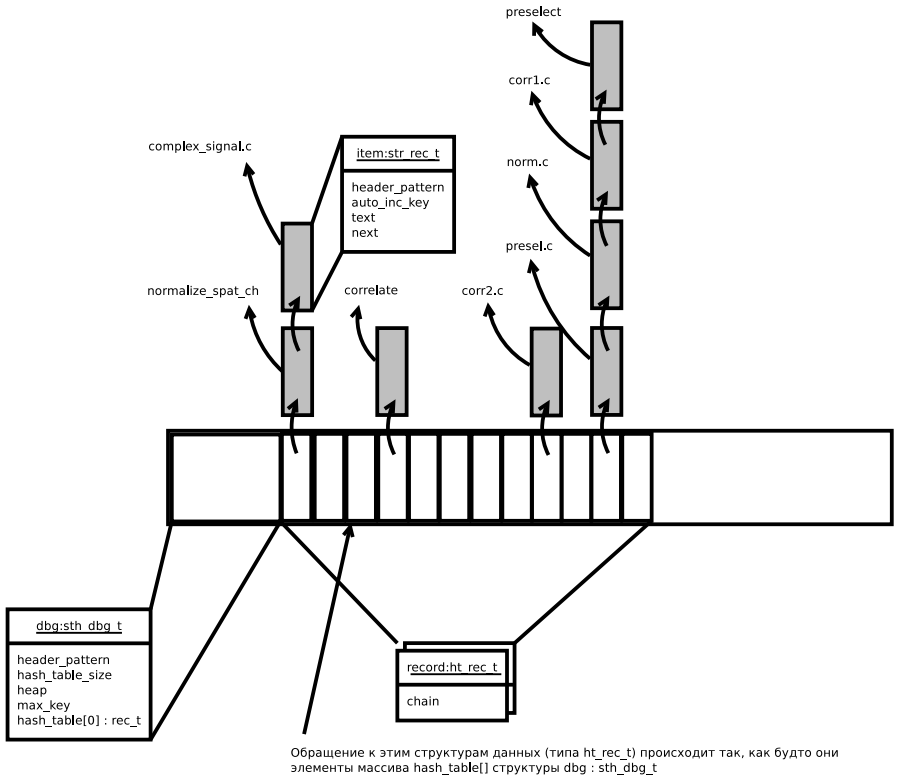

Рис. 4: Устройство хэш-таблицы

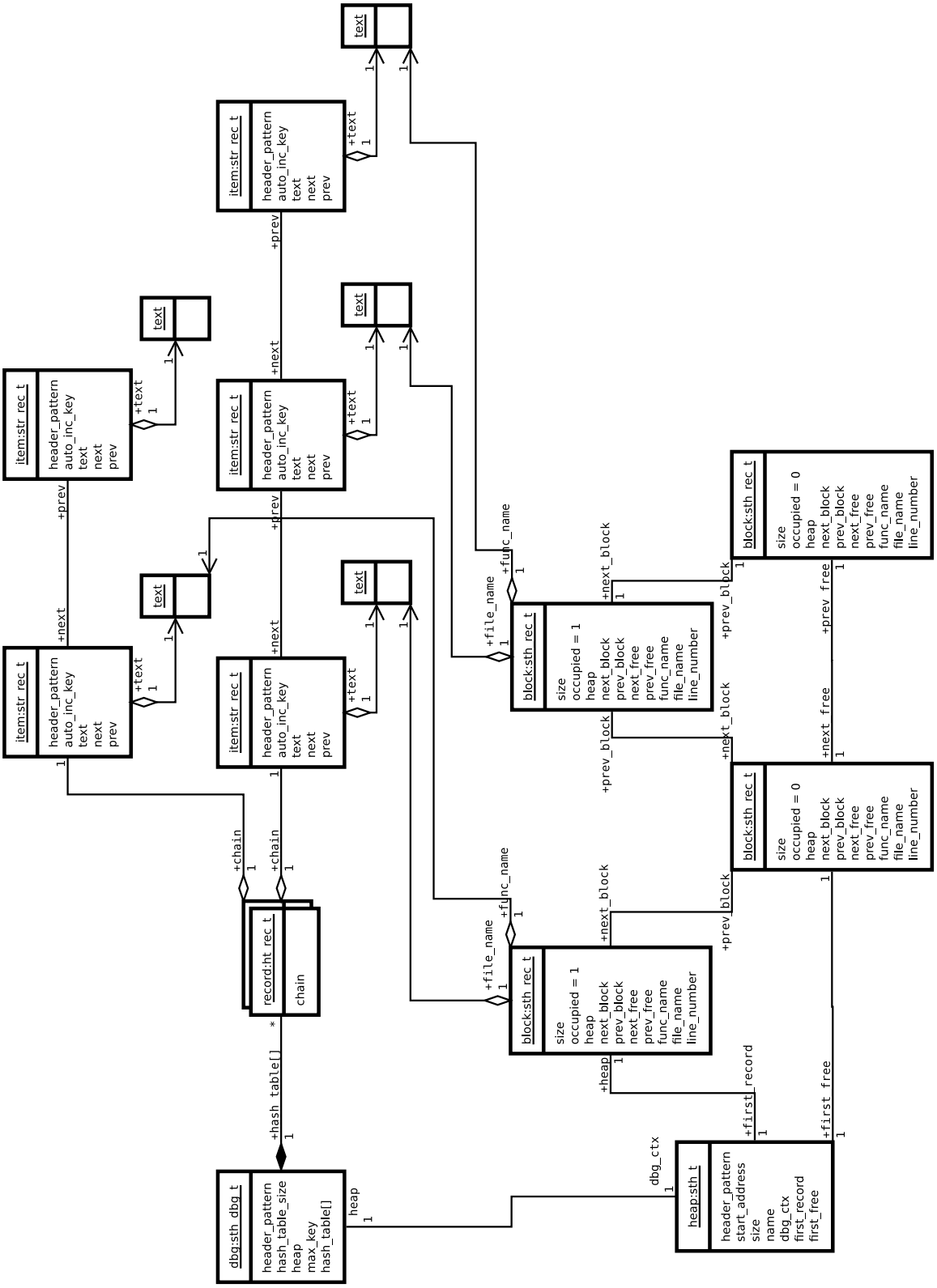

Рис. 5: Диаграмма объектов

7 Лист изменений документа

Дата	Фамилия И.О.	Примечание
17 декабря 2012	Миннигалиев Т.А.	Начальная ревизия