Внутри менеджера памяти. Выбор, подходы и реализация методов динамического выделения памяти

Ниже рассматриваются методики управления памятью, доступные программистам под Linux™. Несмотря на то, то примеры приведены на языке C, все это может применяться и в других языках. В статье рассказывается, как работает механизм управления памятью, и показывается, как вручную управлять памятью, как работать в режиме полу-ручного управления на основе управления референтными счетчиками и динамической областью (пулом), и как управлять памятью в полностью автоматическом режиме, используя сборщик мусора.

Зачем нужно управлять памятью

Управление памятью это одна из самых важных (фундаментальных) областей в компьютерном программировании. Во многих скриптовых языках, вам не нужно заботиться об управлении памятью (так как за вас это делает движок языка), но это не делает проблему управления памятью менее важной. Знания возможностей и ограничений вашего менеджера памяти критически важно для эффективного программирования. В большинстве системных языков (например, C и C++), вы должны самостоятельно заботиться об управлении памятью. Ниже рассмотрены основы автоматического, полуавтоматического и ручного управления памятью.

Возвращаясь во времена ассемблеров для Apple II, можно вспомнить, что управление памятью не вызвало такой большой проблемы. Вы в основном использовали ресурсы все системы. Все, что было доступно машине, то было доступно и вам. Вам даже не приходилось беспокоиться о том, сколько памяти доступно, так как все машины тогда имели одинаковый объем этого ресурса. А так как ресурсы памяти были в основном статическими, то вы просто выбирали область памяти и использовали ее.

Тем не менее, даже на таких простых компьютерах у вас все же возникали некоторые вопросы – особенно, если вы не знали заранее, сколько памяти потребуется каждой из частей вашей программы. Если вы хотите ограничить или изменить требуемый объем памяти, то вам придется столкнуться со следующими действиями:

- Определить достаточно ли у вас памяти для обработки данных
- Получить секцию памяти из доступной памяти
- Вернуть эту секцию обратно в пул доступной памяти, чтобы потом ее могли использовать другие программы.

Библиотеки, которые реализуют функции для удовлетворения этих требований называются *аллокаторами (allocators)*. Название произошло от их основных функций – выделение *(allocating)* и освобождение (*deallocating*) памяти. Чем более динамичная программа (в плане работы с ресурсами), тем больше внимания уделяется вопросу управления памятью, и тем ответственнее становится выбора аллокатора. Давайте взглянем на различные методы доступные для управления памятью, их плюсы и минусы, и ситуации, в которых лучше использовать каждый из них.

C-подобные аллокаторы памяти

Язык программирования C предоставляет две функции для выполнения наших условий:

- **malloc:** Выделяет заданное число байт и возвращает указатель на них. Если доступной памяти не достаточно, что возвращается нулевой (null) указатель.
- **free:** Принимает (в качестве входного параметра) указатель на сегмент памяти выделенный функцией `malloc`, и возвращает его для дальнейшего использования программатором операционной системы (на самом деле, некоторые реализации `malloc` могут вернуть память назад программе, но не операционной системе.).

Физическая и виртуальная память

Чтобы понять, как захватывается память внутри вашей программы, первое, с чем нужно разобраться – это понять, как выделяется память для вашей программы со стороны операционной системы (ведь единоличный хозяин всех ресурсов в системе именно она, и только, она решает, как и кому сдать в аренду часть своего хозяйства). Каждый процесс на вашем компьютере думает, что он имеет доступ ко всей вашей физической памяти. Очевидно, что раз вы запускаете много программ в одно и тоже время, то каждый процесс не может владеть всей памятью. То, что происходит на самом деле называется *виртуальной памятью* (*virtual memory*).

Например, пусть ваша программа имеет доступ к памяти по адресу 629. Система виртуальной памяти, не обязательно содержит её (эту часть памяти) в ОЗУ (RAM) в месте с адресом 629. Эта ячейка памяти может быть вообще не загружена в ОЗУ – она могла быть перемещена на диск, если ваш физический объем ОЗУ был полностью заполнен. Так как адресация не обязательно отражает физическое расположение, где находится память, то при этом и применяется термин виртуальная память. Операционная система имеет таблицу трансляции виртуальных адресов в физические, так что “железо” может корректно отвечать на запрашиваемые адреса. И если запрашиваемы адреса находится на диске, а не в ОЗУ, то операционная система временно приостановит ваш процесс, выгрузит другой участок памяти на диск и загрузит запрошенную память с диска, после чего возобновит ваш процесс. При таком подходе, каждый процесс получает свое собственное адресное пространство для работы и может получить даже больше памяти, чем физически установлено на машине.

На 32-битных системах с архитектурой x86, каждый процесс может адресовать до 4 Гб памяти. Сейчас большинство людей не имеют на своих машинах память в размере 4 Гб, даже если вы включите своп, то *на процесс* должно приходится не более 4Гб. Поэтому когда загружается процесс, то ему выделяется начальный объем памяти до определенного адреса (адресация растет вверх), называемого *system break (системная граница)*. За рамками этой памяти идет так называемая *неразмеченная память (unmapped memory)* – память, для которой не заданны соответствующие физические адреса на диске или в ОЗУ. Поэтому, если процесс намерен за рамки своей инициализированной памяти, то он должен запросить у операционной системы “разметить” еще память (в рамках своего процесса, разумеется). (Разметка (Mapping) это математический термин для определения однозначного (один-к-одному) соответствия – память “размечена” ("mapped"), когда ее виртуальные адреса соответствуют физическим адресам хранящим эту память.)

UNIX-подобные системы имеют два основных системных вызова, которые размечают дополнительную память:

- **brk:** `brk()` – это очень простой системный вызов. Помните выше мы говорили о системной границе (*system break*), месте, где находится граница размеченной памяти процесса? Так вот, `brk()` просто сдвигает эту границу (в сторону уменьшения или увеличения), для добавления или удаления памяти процесса.

- **mmap:** функция `mmap()`, или "memory map," (дословно – карта памяти) похожа на `brk()`, но она гораздо более гибкая. Во-первых, она может разметить память где угодно, а не только в “хвосте” памяти процесса. Во-вторых, она не только может соотнести виртуальные адреса и физические для ОЗУ или свопа, но так же может сделать тоже самое и для файлов и их расположений, так что адресация памяти при чтении и записи будет иметь эффект записи и чтения данных напрямую из файла. Здесь нас интересуют только возможности `mmap's` для добавления разметки ОЗУ для нашего процесса. Имеется также функция `munmap()`, которая по своему действию обратна функции `mmap()`.

Как вы видите и `brk()` и `mmap()` могут использоваться для добавления дополнительной виртуальной памяти к нашему процессу. Мы в наших примерах будем использовать `brk()`, потому что он проще и более распространен.

Реализация простого аллокатора

Если у вас уже есть солидный опыт программирования на C, то вы, скорее всего уже сталкивались с использованием `malloc()` и `free()`. Но, вероятно, вы не задумывались о том, как реализованы эти функции в вашей операционной системе. В этом разделе вы увидите пример простой реализации `malloc` и `free`, который продемонстрирует аспекты, связанные с управлением памятью.

Для того чтобы использовать данные примеры, скопируйте представленные листинги кода и вставьте их в файл с названием malloc.c. Все разъяснения листинга будут приведены ниже. Выделение памяти в большинстве операционных систем управляется двумя простыми функциями:

- `void *malloc(long numbytes)`: Выделяет заданное число байтов памяти и возвращает указатель на первый байт.
- `void free(void *firstbyte)`: Используя указатель, возвращаемый предыдущей функцией `malloc`, она возвращает пространство, которое было выделено обратно в "свободного пространства" процесса.

`malloc_init`

будет нашей функцией для инициализации нашего аллокатора памяти. Она будет делать 3 вещи: помечать наш аллкотор как инициализированный, находить последний валидный адрес памяти в системе (тот который действителен и является допустимым), и устанавливать указатель на начало нашей управляемой памяти. Эти три переменные являются глобальными:

Листинг 1. Глобальные переменные нашего простого аллокатора

```
int has_initialized = 0;

void *managed_memory_start;

void *last_valid_address;
```

Как уже было сказано выше, граница размеченной памяти – последний валидный адрес – известный так же как системная граница (system break) или *текущая граница (current break*). Во многих UNIX® системах, для нахождения текущей системной границы используется функция `sbrk(0)`. `sbrk` смещает текущую системную границу на число байт, указанных в аргументе функции, после чего возвращает новую системную границу. Её вызов с аргументом `0` просто возвращает текущую границу. Вот наш инициализирующий код для `malloc`, который находит текущую границу и инициализирует наши переменные:

Листинг 2. Функция инициализации аллокатора

```
/* Подключаем функцию sbrk */

#include <unistd.h>

void malloc_init()

{

/* Вытаскиваем последний валидный адресс из ОС */

last_valid_address = sbrk(0);

/* Пока у нас еще нет памяти для управления, поэтому
просто устанавливаем начало на last_valid_address

*/

managed_memory_start = last_valid_address;

/* мы все проинициализировали и готовы продолжать */

has_initialized = 1;

}
```

Теперь, чтобы должным образом управлять памятью, нам нужно иметь возможность следить за тем, что мы выделяем и освобождаем. Нам нужно каким-то образом помечать блоки как неиспользуемые после того, как для них будет вызвана `free`, и также определить блоки как используемые при вызове `malloc`. Значит, начало каждого кусочка памяти `malloc`'ом должно быть запомнено в следующей структуре:

Листинг 3. Определение структуры Блока Управления Памятью (Memory Control Block)

```
struct mem_control_block {

int is_available;
```

```
int size;

};
```

Теперь, вы наверное думаете, что вызов `malloc` может создать проблемы для программ - как они узнают об этой структуре? Ответ прост – они о ней и не должны знать; мы её спрячем, передвинув указатель назад на размер этой структуры, перед тем как его вернуть. Это заставит вернуть указатель на память, которая не используется ни для каких других целей. Таким образом, с точки зрения вызывающей программы, все что она получает – свободная доступная память. Потом, когда указатель возвращается назад через `free()`, мы просто восстанавливаем несколько байт памяти, чтобы снова найти эту структуру.

Мы должны немножко поговорить об освобождении ресурсов, прежде чем перейдем к рассмотрению захвата памяти, потому что это несколько проще. Единственная вещь, которую нам нужно сделать для освобождения памяти – это взять указатель, который мы сами и выдали, восстановить байты структуры `sizeof(struct mem_control_block)`, и пометить их свободные. Вот код, который выполняет это:

Листинг 4. Функция освобождения памяти

```
void free(void *firstbyte) {

struct mem_control_block *mcb;

/* По полученному указателю находим mem_control_block
и восстанавливаем структуру

*/

mcb = firstbyte - sizeof(struct mem_control_block);

/* Помечаем блок как доступный (свободный) */

mcb->is_available = 1;

/* Вот и все! Мы закончили. */

return;

}
```

Как вы наверное заметили, в этом аллокаторе, освобождение памяти занимает постоянное время, и используется очень простой механизм. А вот захват памяти немного сложнее. Вот схема алгоритма:

Листинг 5. Псевдо-код для главного аллокатора (main allocator)

1. Если наш аллокатор не был инициализирован, то инициализируем его.

2. Добавляем к запрашиваемому размеру размер нашей структуры sizeof(struct mem_control_block).

3. Запускаем managed_memory_start.

4. Мы находимся на last_valid адресе?

5. Если да:

A. Значит мы не нашли ни одного куска памяти, который был бы достаточно большим.

-- запрашиваем у операционной системы еще памяти и возвращаем ее.

6. В противном случае:

A. Доступна ли текущая область (запускаем проверку is_available из the mem_control_block)?

B. Если проверка выполнена успешно:

i) Достаточно ли большая это область (проверяем "size" в mem_control_block)?

ii) Если да:

a. Помечаем ее как недоступную (занятую)

b. Откатываем указатель на величину mem_control_block и возвращаем указатель

iii) В противном случае:

a. Сдвигаем границу на "size" байт

b. Переходим к пункту 4

C. В противном случае:

i) Сдвигаем границу на "size" байт

```
                    ii) Переходим к пункту 4
```

Мы в основном работаем с памятью используя связанные указатели, ища открытые куски памяти. Вот код:

Листинг 6. Главный аллокатор

```
void *malloc(long numbytes) {

        /* Запоминаем текущую область памяти, где мы
        ищем */

        void *current_location;

        /* Это тоже самое, что и current_location,
        но располагается выше на величину
        memory_control_block */

        struct mem_control_block
        *current_location_mcb;

        /* Это место памяти мы вернем. Оно будет
        установлено в 0, пока мы не найдем
        подходящего размера памяти в доступной
        области */

        void *memory_location;

        /* Инициализируемся, если мы еще этого не
        сделали */

        if(! has_initialized) {

                malloc_init();

                }

        /* Память, которую мы ищем, должна включать
        память для хранения блока управления памятью
        (memory control block), но пользователям
        функции malloc не обязательно знать об этом,
        поэтому мы просто скрываем это от них.*/

        numbytes = numbytes + sizeof(struct
        mem_control_block);
```

```
/* Устанавливаем значение memory_location в
0, пока мы не найдем подходящего места
(куска в памяти)*/

memory_location = 0;

/* Начинаем искать с начала управляемой
памяти. */

current_location = managed_memory_start;

/* Продолжаем это делать, пока не просмотрим
все доступное пространство */

while(current_location !=
last_valid_address)

     {

     /*          current_location          и
     current_location_mcb указывают на один
     и    тот    же    адрес.    Так    как
     current_location_mcb              имеет
     соответствующий тип, то мы можем его
     использовать в качестве структуры.
     current_location     это     свободный
     указатель (void pointer), поэтому мы
     можем    его    использовать,    чтобы
     просчитать адресацию. */

     current_location_mcb =

                    (struct mem_control_block
                    *)current_location;

     if(current_location_mcb->is_available)

          {

          if(current_location_mcb->size >=
          numbytes)

               {

               /* Ура! Мы нашли открытое (не
               занятое) место подходящего
               размера.*/

               /* Упс, теперь оно больше не
               свободно
```

```
J*/

current_location_mcb->is_available = 0;

/* Теперь мы его владельцы */

memory_location = current_location;

/* Выходим из цикла */

break;

}

}

/* Так как Текущий блок памяти не подходит, то переходим дальше
*/

current_location = current_location +

               current_location_mcb->size;

}

/* Если нам все еще не удалось найти подходящее место в памяти,
то мы просим у операционной системы еще памяти */

if(! memory_location)

     {

     /* Сдвигаем границу на заданное число байт вверх */

     sbrk(numbytes);

     /* Новая память будет находится в месте, где раньше
     находился последний валидный адрес (после выделения новой
     памяти он сместился). */

     memory_location = last_valid_address;

     /* Мы сдвигаем значение последнего валидного адреса на
     numbytes байт. */

     last_valid_address = last_valid_address + numbytes;

     /* Теперь нужно инициализировать mem_control_block */

     current_location_mcb = memory_location;
```

```
    current_location_mcb->is_available = 0;

    current_location_mcb->size = numbytes;

    }

/* Теперь, в любом случае (ну за исключением, состояния ошибки),
memory_location адресует память, включая блок mem_control_block
*/

/* Передвигаем указатель назад на размер структуры
mem_control_block */

memory_location = memory_location + sizeof(struct
mem_control_block);

/* Возвращаем указатель */

return memory_location;

}
```

Ну вот он, наш менеджер памяти. Нам осталось только все это собрать и заставить работать с нашими программами. Для того, чтобы собрать ваш `malloc`-совместимый аллокатор (вообще-то, мы опустили здесь некоторые функции типа `realloc()`, но `malloc()` и `free()` являются основными), запустите следующую команду:

Листинг 7. Компиляция аллокатора

```
gcc -shared -fpic malloc.c -o malloc.so
```

В результате выполнения этой команды вы получите файл с именем `malloc.so`, который является библиотекой общего пользования, содержащей наш код.

На UNIX системах, для использования вашего аллокатора вместо системного `malloc()` нужно сделать следующее:

Листинг 8. Замещение вашего стандартного malloc'a

```
LD_PRELOAD=/path/to/malloc.so
```

```
export LD_PRELOAD
```

Переменная окружения LD_PRELOAD, заставит динамический компановщик загрузить данные указанной расшаренной библиотеки до того, как его загрузит какой-либо выполняющийся код. Это также даст преимущество (повысит приоритет) данных указанной библиотеки. Так что, теперь любое запущенное нами в течении этой сессии приложение будет использовать наш `malloc()` вместо системного. Существуют некоторые приложения, которые не используют `malloc()`,но это редкость. Другие, которые используют другие функции управления памятью, такие как `realloc()` или, которые не обременяют себя заботой о внутреннем поведении `malloc()` скорее всего, просто не будут работать (может быть просто вылетят в "корку"). Интерпретор `ash shell`, демонстрирует отличную работу с использованием нашего нового `malloc()`.

Если вы хотите удостовериться, что используется ваш `malloc`(), то вы должны использовать вызов `write`() в точках входа ваших функций.

Конечно, наш менеджер памяти еще не реализует многое из того, что надо бы, но он все же хорошо демонстрирует, что требуется для работы менеджера памяти. Вот его некоторые недостатки:

- Так как он оперирует с системной границей (глобальная переменная), то он не может сосуществовать с другим аллокатором или с `mmap`.
- Во время захвата памяти, в худшем случае, он пробежится через *все* процессы в памяти; а это может занять достаточно много времени – т.к. сюда также включается и память доступная на диске, что означает, что операционная система будет тратить время на пересылку данных с диска и обратно.
- Здесь нет приличного обработчика ошибок связанных с выходом за рамки зарезервированной памяти (`malloc` просто псевдо-удачно завершается).
- Он не реализует никаких дополнительных функций для работы с памятью, таких как `realloc()`.
- Так как `sbrk()` может вернуть больше памяти, чем мы запросили, то возможны утечки памяти.
- Флаг `is_available` использует 4-байтное слово, даже при том, что он содержит всего лишь 1 бит информации.
- Аллокатор не "потоко-безопасен".
- Аллокатор не может объединять свободные участки в большие блоки.
- Слишком простой алгоритм ведет к высокой потенциальной фрагментации памяти.
- Наверняка есть еще и много других проблем. Поэтому это только пример!

Другие реализации malloc

Существуют много реализации `malloc()`, каждая из них имеет свои сильные и слабые стороны. Есть несколько неоднозначных моментов, которые вам придется принять во внимание во время проектирования вашего собственного аллокатора, включая:

- Скорость выделения памяти
- Скорость освобождения памяти
- Поведение в условиях потока (threaded environment)
- Поведение, когда память почти закончилась
- Расположение Кэша
- Переполнение памяти с управляющими данными
- Роведения в условиях виртуальной памяти (Virtual Memory Environments)
- Малые и большие объекты
- Поддержка режима реального времени

В нашем простом аллокаторе выделение памяти происходи очень медленно, зато освобождение – очень, очень быстро. Так же из-за не проработки поведения в условиях систем виртуальной памяти, наша реализация лучше работает с большими объектами.

Существуют много других реализаций аллокаторов. Среди них:

- **Doug Lea Malloc:** Doug Lea Malloc на самом деле целое семейство аллокаторов, включая оригинальный аллокатор Doug Lea, который является аллокатором лицензии GNU libc, и `ptmalloc`. Doug Lea аллокатор имеет базовую структуру очень схожую с нашей, но он использует индексирование для ускорения процесса поиска и поддерживает возможность объединения нескольких не используемых областей в одну большего размера. Он также позволяет кэшировать, чтобы снова использовать освобождаемую память (что значительно ускоряет процесс). `ptmalloc` - это версия Doug Lea Malloc, которая расширена за счет поддержки многопоточности.
- **BSD Malloc:** BSD Malloc, реализация, которая была разработана для 4.2 BSD и входит в состав FreeBSD. Этот аллокатор, выделяет память под объекты из пула объектов с заранее определенным размером. У него есть классы размеров для объектов различных размеров. Эти размеры имеют величину степени двойки минус константа. Поэтому, если вы запросите объект заданного размера, то он просто выдаст ему (разместит его) в любом классе, которые сможет вместить объект. Это пример быстрой реализации, но может привести к растрате памяти.
- **Hoard:** Hoard был написан с целью создания очень быстрого аллокатора в условиях многопоточной среды. Поэтому, структура составлена таким образом, чтобы наилучшим образом использовать механизм блокировок и при этом уменьшит время ожидания выделения памяти для процессов. Он может серьезно увеличить скорость для многопоточных процессов, которые активно работают с захватом и освобождением памяти.

Эти аллокаторы наиболее известны из всех доступных аллокаторов. Если для вашей программы требуется специфический аллокатор, то вы, возможно, предпочтете сами написать его, чтобы он лучше удовлетворял требования вашей программы. С другой стороны, если вы не очень хорошо знакомы с проектированием аллокаторов, то написание собственного аллокатора может создать больше проблем, чем принести пользы. В качестве хорошего введения в курс можно посоветовать книгу Дональда Кнута *The Art of Computer Programming Volume 1: Fundamental Algorithms* раздел 2.5, "Dynamic Storage Allocation" (Искусство Программирования, том первый, раздел 2.5 “Динамическое Выделение Памяти”).

Оно не много расплывчато, так как там нет деталей сред виртуальной памяти, но большинство базовых алгоритмов здесь изложено.

В C++, вы можете создать свой аллокатор на базе класса или шаблона, перегрузив оператор `new()`. В книге Andrei Alexandrescu's *Modern C++ Design,* описывается аллокатор маленьких объектов в Главе 4, "Small Object Allocation".

Изъяны malloc()- подобных менеджеров памяти

Не только наш менеджер памяти имеет недостатки, есть много недостатков `malloc()`-подобных менеджеров памяти, которые остаются, не зависимо от того, каким аллокатором вы пользуетесь. Управление памятью с помощью `malloc()` может существенно усложнить жизнь программам, которые имеют долго-используемые хранилища, с которыми они работают. Если у вас имеется множество ссылок на память, то бывает сложно отследить момент, когда она должна быть освобождена. Память, чьё время жизни ограничено текущей функцией достаточно просто управлять, но для памяти, которая живет независимо, это представляет значительную трудность. Так же для многих API не понятно, лежит ли ответственность за управлением памятью на вызывающей программе или вызванной функции.

Из-за проблем управления памятью, многие программы ориентированы на правила управления памятью. Обработчики исключительных ситуаций в C++ еще сильнее обостряют эту проблему. Порой кажется, что большинство кода направлено на управление захватом и освобождением памяти, чем на решение непосредственной задачи! Поэтому, мы рассмотрим еще некоторые альтернативные методы управления памятью.

Стратегии Полуавтоматического управления памятью

Референтные счетчики (референтрая целостность)

Референтная целостность - это полуавтоматический метод управления памятью, который подразумевает некоторую поддержку со стороны программиста, но не требует от вас точного знания, когда объект больше не будет использоваться. За это будет отвечать механизм референтной целостности.

В случае с референтной целостностью, все общие(расшаренные) структуры данных имеют поле, которое содержит число активных ссылок ("references") на эту структуру. Когда процедура передают указатель на структуру данных, то референтный счетчик (счетчик ссылок) увеличивается. Таким образом, вы сообщаете структуре данных, в скольких местах она расположена. Потом, когда процедура закончит использовать эти данные, она уменьшит значение счетчика. В этот момент так же проверяется, не равен ли референтный счетчик нулю. Если да, то происходит освобождение памяти.

Преимуществом такого подхода является то, что вам теперь не нужно в точности воспроизводить весь путь до структуры (вы просто создаете еще один указатель и инициализируете его уже известным значением). Каждая новая ссылка просто увеличивает или уменьшает значение счетчика. Это предотвратит освобождение памяти, пока она еще используется. Главное не забывать использовать функцию референтного счетчика, когда вы намерены работать с референтно-отслеживаемыми структурами данных. Так же встроенные функции и библиотеки от сторонних производителей ничего не будут знать о вашем механизме референтной целостности (и соответственно не будут им пользоваться). Референтные счетчики также имеют некоторые проблемы, связанные с образованием кольцевых ссылок на структуры (т.е. когда две структуры ссылаются друг на друга).

Для реализации референтной целостности, вам просто нужны три функции -- одна для увеличения референтного счетчика, а другая – для уменьшения, и еще одна для освобождения памяти, когда референтный счетчик станет равным нулю.

Вот пример функции референтной целостности:

Листинг 9. Базовые функции референтного счетчика

```
/* Определение Структуры */

/* Базовая структура содержит референтный счетчик
refcount */

struct refcountedstruct

     {

     int refcount;

     }

/* Все структуры с refcounted должны отображать
структуру refcountedstruct для своих первых переменных
*/

/* Функции для работы с refcount */

/* Увеличение значения reference */

void REF(void *data)

     {

     struct refcountedstruct *rstruct;

     rstruct = (struct refcountedstruct *) data;

     rstruct->refcount++;

     }

/* Уменьшение значения reference */

void UNREF(void *data)

     {

     struct refcountedstruct *rstruct;

     rstruct = (struct refcountedstruct *) data;

     rstruct->refcount--;
```

```
        /* Освободить эту структуру, если ее больше никто
        не использует */

        if(rstruct->refcount == 0)

                {

                free(rstruct);

                }

        }
```

REF и UNREF

могут иметь более сложную реализацию в зависимости от того, чего вы хотите добиться. Например, возможно вы захотите добавить блокировку для многопоточной программы, и вы можете расширить `refcountedstruct`, чтобы она также включала указатель на функцию для вызова ее до непосредственного освобождения памяти (как деструкторы в объектно-ориентированных языках -- это может потребоваться, если ваша структура будет содержать указатели).

Используя REF и UNREF, вы должны придерживаться следующих правил при присвоении указателей:

- UNREF значение указателя слева указывает до операции присвоения.
- REF значение, которое находится слева от указателя указывает на результат присвоения (пост факт).

  В функциях, которые получают в качестве параметра структуру refcounted, действуют следующие правила :

- REF любой указатель в начале функции.
- UNREF любой указатель в конце функции.

Небольшой пример использования референтной целостности:

Листинг 10. Пример использования reference counting

```
        /* ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ */

        /* Тип данных, которые будут отслеживаться счетчиком
        */

        struct mydata

        {
```

```
        int refcount; /* тоже самое что и refcountedstruct */

        int datafield1; /* Специфические поля данной структуры
        */

        int datafield2;

        /* здесь могут находиться другие необходимые описатели
        */

        };

        /* Использование функций в коде */

        void dosomething(struct mydata *data)

        {

        REF(data);

        /* Обработка данных */

        /* что-то делаем */

        UNREF(data);

        }

        struct mydata *globalvar1;

        /* Заметьте, здесь мы не увеличиваем счетчик, так как
        мы сохраняем указатель в глобальной переменной до
        конца обработки вызова функции */

        void storesomething(struct mydata *data)

        {

        REF(data); /* передаем как параметр */

        globalvar1 = data;

        REF(data); /* используем ref, так как выполнена
        операция присваивания */

        UNREF(data); /* функция закончена */

        }
```

Так как механизм референтных счетчиков достаточно прост, то многие программисты используют собственные реализации вместо стандартных библиотек. Хотя в итоге работа все же сводится к использованию все тех же низкоуровневых аллокаторов как `malloc` и `free` для фактического захвата и освобождения памяти.

Механизм референтных счетчиков так же используется иногда и в высокоуровневых языках, таких как Perl, для реализации менеджеров памяти. В этих языках, этот механизм контролируется автоматически средствами языка, поэтому вам не нужно беспокоиться почти ни о чем, кроме написания модулей расширения. Это, конечно, ведет к небольшому снижению скорости, так как все ссылки должны быть посчитаны, но зато это дает некоторые преимущества в виде безопасности и легкости программирования. Вот его преимущества:

- Простая реализация.
- Легкость использования.
- Так как ссылка является частью структуры данных, то легко производить кэширование.

Но есть и недостатки:

- Вы никогда не должны забывать о вызове функций изменения состояния счетчика.
- При образовании циклических структур, они никогда не будут освобождены.
- Он немного “подтормаживает” во время каждой процедуры присваивания указателя.
- Вы должны проявлять особую предосторожность при использовании обработчиков ошибок (таких как `try` или `setjmp()/longjmp()`), во время использования объектов в референтными счетчиками.
- Требуется дополнительная память для работы с ссылками.
- Референтный счетчик захватывает первую позицию в структуре, которая на большинстве машин является самой быстрой в плане доступа.
- Он работает медленней и сложнее в реализации для случая многопоточной среды.

C++ может смягчить некоторые из ошибок программиста при использовании *smart pointers (умных указателей)*, которые могут работать с деталями обработчиков-указателей, например, выполнять для вас подсчет ссылок. С другой стороны, если вы используете какой-либо наследуемый код, которые не может обработать ваши умные (smart pointers) (например, линковка с библиотекой C), то это, скорее всего, превратит все ваши хорошие начинания в кашу и сильно запутает ситуацию. Поэтому, это полезно только для “чистых” C++ проектов. Больше об “умных указателях” можно прочитать в книге Alexandrescu's *Modern C++ Design* в главе "Smart Pointers".

Пулы памяти

Пулы памяти – это другой метод полуавтоматического управления памятью. Пулы памяти помогают автоматизировать управление памятью для программ, которые проходят определенные стадии, на каждой из которых используется память, которая выделяется только на время обработки данной стадии. Например, множество процессов сетевого сервера захватывают много памяти для нужд пре-коннекции – это память, чьё максимальное время жизни ограничено текущим соединением. Сервер Апач, который использует пуловую память, работает с соединениями разбивая “жизнь” их на стадии, и на каждой имеется собственный пул памяти. В конце стадии, существующий пул очищается.

В пуловом управлении памятью каждый захват ресурса определяет пул памяти, в котором

будет производится выделение памяти. Каждый пул имеет свое время жизни. В Апаче есть пул, время жизни которого равно времени жизни сервера, а есть и такой, у которого время жизни определяется временем жизни соединения, есть еще и со временем жизни равным времени жизни запроса и т.д. Поэтому, он имеет целую серию функций, которые не будут генерировать каких либо данных со временем жизни превышающим время соединения. В добавлении к этому, некоторые реализации позволяют использовать вычищающие функции (*cleanup functions)*, которые вызываются перед непосредственным очищением пула, для выполнения разнообразных задач (аналогично деструкторам для объектно-ориентированных языков).

Для использования пулов в ваших программах, вы можете использовать реализацию из библиотеки GNU или использовать Apache's Apache Portable Runtime. Вариант с GNU предпочтительнее, потому что он входит в стандартную поставку Linux. А Apache Portable Runtime преимущество заключается в наличии множества утилит для работы с различными аспектами при работе мультиплатформенным серверным программным обеспечением.

Вот пример гипотетического кода, который демонстрирует использования obstacks:

Листинг 11. Пример кода для obstacks

```
#include <obstack.h>

#include <stdlib.h>

     /* пример кода при использовании obstacks */

     /* Для obstack используем макрос (xmalloc это
     функция malloc, которая закончит работу, если
     память закончится */

     #define obstack_chunk_alloc xmalloc

     #define obstack_chunk_free free

     /* Пулы */

     /* Сюда должны попадать только долговременные
     аллокаторы */

     struct obstack *global_pool;

     /* Это пул для данных пре-коннекции */

     struct obstack *connection_pool;

     /* Это пул для данных пре-запроса */

     struct obstack *request_pool;

     void allocation_failed()
```

```
                    {
                    exit(1);
                    }
          int main()
                    {
                    /* Инициализируем пулы */
                    global_pool = (struct obstack *)
                    xmalloc (sizeof (struct obstack));
                    obstack_init(global_pool);
                    connection_pool = (struct obstack *)
                    xmalloc (sizeof (struct obstack));
                    obstack_init(connection_pool);
                    request_pool = (struct obstack *)
                    xmalloc (sizeof (struct obstack));
                    obstack_init(request_pool);
                    /* Устанавливаем функцию обработки
                    ошибок */
                    obstack_alloc_failed_handler =
                    &allocation_failed;
                    /* Главный цикл сервера */
                         while(1)
                              {
                              wait_for_connection();
                              /* Для установленного
                              соединения */
                              while(more_requests_available(
                              ))
```

```
                                        {

                                        /* Обрабатываем
                                        запрос */

                                        handle_request();

                                        /* Освобождаем все
                                        память, захваченную
                                        в пуле запросов */

                                        obstack_free(request
                                        _pool, NULL);

                                        }

                                /* Мы закончили работу с
                                соединением, время освободить
                                пул */

                                obstack_free(connection_pool,
                                NULL);

                                }

                        }

                int handle_request()

                        {

                        /* Убеждаемся, что все аллокаторы
                        объектов захватывали память в пуле
                        запросов */

                        int bytes_i_need = 400;

                        void *data1 =
                        obstack_alloc(request_pool,
                        bytes_i_need);

                        /* Обрабатываем запрос и др. */

                        /* выходим */

                        return 0;

                        }
```

Обычно, после каждой важной стадии операции временный стек (`obstack`) этой стадии

освобождается. Заметьте, что если процедуре нужно захватить память, которая должна существовать дольше, чем длится данная стадия, то она может использовать так называемый стек длительного хранения (`longer-term obstack`), такой как используемый для соединений или глобальный. Если в `obstack_free()` передается значение NULL, то это значит, что временный стек должен быть очищен от всего содержимого. Доступны так же и другие значения, но они не так полезны.

Преимущества использования пулового метода захвата памяти:

- Для приложения так проще управлять памятью.
- Выделение и освобождение памяти гораздо быстрее, так как действие производится единовременно и с целым пулом. Захват памяти может быть осуществлен за время O(1), и освобождение пула потребует примерно столько же.
- Пулы с обработчиками ошибок могут использовать механизм предварительного захвата, что позволяет вашей программе продолжать работу, даже если закончится доступная память.
- Существуют стандартные реализации, которые просты в использовании.

Недостатки метода:

- Пулы памяти полезны только для программ, у которых четко выделены стадии обработки.
- Часто пулы памяти не очень хорошо работают для библиотек от сторонних разработчиков.
- Если структура программы изменилась, то возможно и пулы должны быть модифицированы, а это приведет к пересмотру всей системы управления памятью.
- Вы должны помнить, в каком пуле производить захват памяти. Если вы произведете захват не в том пуле, то такую ошибку будет трудно отследить.

Сборщик мусора

Сборщик мусора (Garbage collection)

- это полностью автоматическое определение и удаление объектов данных, которые не используются. Сборщики мусора в основном запускаются, когда доступная память преодолевает определенный порог. Обычно они начинают свою работу с некоторого адреса “базы данных”, которая доступна программе - стек данных, глобальные переменные и регистры. После чего проверяется каждый кусочек данных связанный с этими адресами. Все что находит сборщик считается “хорошими” данными; если связанная память не найдена, то эти адреса считаются мусором и они могут быть уничтожены и использованы заново. Для эффективного управления памятью многие типы сборщиков требуют знания расположения указателей в структурах данных, и поэтому для правильного функционирования встраиваются прямо в языки программирования.

Типы сборщиков

- **Copying (Копировщики):**

При этом хранилище памяти делится на две части и данные существуют только на одной из них. Периодически, производится копирование данных из одной части в другую начиная с "базовых" элементов. Теперь новая занятая секция памяти становится активной, а все оставшееся на другой части считается мусором. Так же, когда происходит это копирование, все указатели обновляют свое значение и указывают на новое положение памяти. Поэтому,

для использования этого метода сбора мусора, сборщик должен быть интегрирован в язык программирования.

- **Mark and sweep (Маркировщик-выметала):** Все кусочек данных помечаются с помощью тегов. Изредка все теги устанавливаются в значение 0, и сборщик пробегается по всем "базовым" элементам. Если встречается рабочая память, то она помечается тегом равным 1. Все, что не помечено тегом равным 1 считается мусором и будет уничтожено.
- **Incremental(Инкрементные):** Инкрементные сборщики мусора не требуют работы со всеми объектами данных. Запуск для проверки всей памяти может привести к проблемам, так в период работы коллектора все останавливается и из-за проблем с кэшированием данных ассоциированных со всеми текущими данными. Инкрементные проблемы позволяют избежать этих проблем.
- **Conservative(Консерваторы):** Консервативным сборщикам мусора не нужно ничего знать о структуре ваших данных для управления памятью. Они просто просматривают все байты данных и делают предположение о том, *могут ли эти данные быть* указателями. Если они находят такой предполагаемый указатель, то память, на которую они указывают, помечается как связанная. Это иногда может приводить к проблемам, когда память, которая на самом деле не является связанной считается таковой. Например, целочисленное поле содержит значение, которое было адресом захваченной памяти. Но это случается достаточно редко, и ведет к не значительной растрате памяти. К достоинствам консервативным сборщиков можно отнести их возможность интегрирования с любым языком программирования.

Консервативные сборщик мусора Hans Boehm один из самых популярных доступных сборщиков мусора, он свободно распространяется и имеет как режим работы инкрементного, так и консервативного сборщика. Вы можете использовать его как временное решение вместо вашего системного аллокатора (используя `malloc/ free` вместо оригинальных API) путем встраивания его директивой `--enable-redirect-malloc`. Таким образом, вы можете осуществить то же самый трюк с `LD_PRELOAD`, который мы использовали в нашем простом аллокаторе, чтобы сделать доступным сборщик практически из любой программы на вашей системе. Если вы подозреваете утечку памяти в программе, то вы можете воспользоваться данным сборщиком предотвращения потери памяти процессом. Многие использовали эту технику во времена ранней Mozilla, когда у неё наблюдались серьезные утечки памяти. Этот сборщик мусора может работать как под Windows® , так и под UNIX.

Некоторые преимущества сборщиков мусора:

- Вам никогда не придется беспокоиться о двойном освобождении памяти или времени жизни объекта.
- С некоторыми сборщиками вы можете использовать те же API, которые вы используете для обычного захвата памяти.

К недостаткам относятся:

- Большинство сборщиков не сообщают, когда будет освобождаться ваша память.
- Во многих случаях, сборщик мусора работает медленнее, чем другие виды менеджеров памяти.
- Ошибки (баги) по причине сборщиков мусора трудно отследить.
- У вас все еще могут быть утечки памяти, если вы забудете установить неиспользуемые указатели в значение `null`.

Заключение

Существует множество образцов менеджеров памяти. Каждый такой образец имеет широкий спектр реализаций со своими достоинствами и недостатками. Используя техники по умолчанию в ваших программных средах годится в большинстве случаев, знание доступных вариантов поможет вам в случае, если вашему приложению потребуются специфические подходы к решению поставленной задачи.