# Менеджмент памяти или как реже стрелять себе в ногу

Привет, Читатель! В этой статье я попытаюсь рассказать, что из себя представляет менеджмент памяти в программах/приложениях с точки зрения прикладного программиста. Это не исчерпывающее руководство или мануал, а просто обзор существующих проблем и некоторых подходов к их решению.

Зачем это необходимо? Программа - это последовательность команд по обработке данных (в самом общем случае). Эти данные необходимо некоторым образом **хранить**, **загружать**, **передавать** и т.д. Все эти операции не происходят мгновенно, следовательно, они непосредственно влияют на скорость работы вашего конечного приложения. Умение оптимально управлять данными в процессе работы позволит вам создавать весьма нетривиальные и очень требовательные к ресурсам программы.

Замечание: основная часть материала излагается с примерами из игр/игровых движков (т.к. эта тема более интересная лично для меня), однако большую часть материала можно применить и к написанию серверов, пользовательских приложений, пакетов графики и т.д.

 Невозможно все держать в памяти. Но если не успел подгрузить, то получишь мыло

# С места в карьер

Так сложилось в индустрии, что крупные ААА игровые проекты разрабатываются преимущественно на движках, написанных с помощью C++. Одна из особенностей этого языка заключается в необходимости ручного управления памятью. Java/C# и т.д. могут похвастаться сборкой мусора (GarbageCollection/GC) — возможностью создавать объекты и при этом не освобождать использованную память руками. Этот процесс упрощает и ускоряет разработку, но может вызвать и некоторые проблемы: периодически срабатывающий сборщик мусора способен убить весь soft-real time и добавить неприятные зависания в игру.

Да, в проектах типа "Minecraft" работа GC может быть и незаметна, т.к. они в целом не требовательны к ресурсам вычислителя, однако такие игры, как "Red Dead Redemption 2", "God of War", "Last of Us", работают "едва ли" не на пике производительности системы и поэтому нуждаются не только в большом количестве ресурсов, но и в грамотном их распределение.

Кроме этого, работая в среде с автоматическим выделением памяти и сборкой мусора, вы можете столкнуться с нехваткой гибкости в управлении ресурсами. Не секрет, что Java скрывает под капотом все детали реализации и аспекты своей работы, поэтому на выходе вы имеете только установленный интерфейс взаимодействия с ресурсами системы, но его может не хватить при решении некоторых задач. Например, запуск в каждом кадре алгоритма с неконстантным количеством аллокаций памяти (это может быть поиск путей для AI, проверка видимости, анимации, и т.д.) неизбежно ведет к катастрофическому падению производительности.

# Как выглядят аллокации в коде

Прежде чем продолжить изложение, я бы хотел показать, как непосредственно происходит работа с памятью в C/C++ на паре примеров. В общем случае, стандартный и самый простой интерфейс для распределения памяти процесса представляется следующими опрециями:

// получить указатель на свободный участок памяти размером size байт  
void\* malloc(size\_t size);   
  
// освободить участок памяти по указателю p  
void free(void\* p);

Здесь можно добавить про дополнительные функции, которые позволяют выделить выровненный участок памяти:

// C11 стандарт - выделить участок памяти по адресу, кратному\* alignment  
void\* aligned\_alloc(size\_t size, size\_t alignment);  
  
// Posix стандат - выделить участок выровненной памяти и положить  
// адрес на него в переменную по указателю address (\*address = allocated\_mem\_p)  
int posix\_memaling(void\*\* address, size\_t alignment, size\_t size);

Обратите внимание на то, что на различных платформах могут поддерживаться различные стандарты функций, доступные, например на macOS, и недоступные на win.

Забегая вперед, выровненные **особым образом** участки памяти могут вам понадобиться как для попадания в кэш-линии процессора, так и для вычислений с помощью расширенного набора регистров ([SSE](https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions), [MMX](https://en.wikipedia.org/wiki/MMX_(instruction_set)), [AVX](https://en.wikipedia.org/wiki/Advanced_Vector_Extensions), и т.д).

Пример "игрушечной" программы, которая выделяет память и печатает значения буфера, интерпретируя их как целые числа со знаком:

/\* main.cpp \*/  
#include <cstdio>  
#include <cstdlib>  
  
int main(int argc, char\*\* argv) {  
const int N = 10;  
int\* buffer = (int\*) malloc(sizeof(int) \* N);  
for(int i = 0; i < N; i++) {  
printf("%i ", buffer[i]);  
}  
free(buffer);  
return 0;  
}

На macOS 10.14 данную программу можно собрать и запустить следующим набором команд:

$ clang++ main.cpp -o main  
$ ./main

Замечание: здесь и далее я не очень хочу освещать такие операции C++ как new/delete, так как они скорее относятся к конструированию/уничтожению непосредственно объектов, но под собой используют обычные операции по работе с памятью наподобие malloc/free.

# Проблемы с памятью

Существует несколько проблем, возникающих при работе с ОЗУ компьютера. Все они, так или иначе, вызваны не только особенностями ОС и ПО, но и архитектурой железа, на котором все это добро работает.

### 1. Количество памяти

К сожалению, память ограничена физически. На PlayStation 4 это 8 GiB GDDR5, [3.5 GiB из которых операционная система резервирует для своих нужд](https://www.eurogamer.net/articles/digitalfoundry-ps3-system-software-memory). Виртуальная память и подкачка страниц особо не помогут, так как swapping страниц на диск - операция весьма медленная (в рамках фиксированных N кадров в секунду, если говорить об играх).

Также стоит отметить и ограниченный "*бюджет*" - некоторое искусственное ограничение на объем используемой памяти, созданное с целью запуска приложения на нескольких платформах. Если вы создаете игру для мобильной платформы и хотите поддержать не одно, а целую линейку устройств, вам придется ограничить свои аппетиты в угоду обеспечения более широкого рынка сбыта. Это может быть достигнуто как просто ограничением расхода ОЗУ, так и возможностью конфигурировать это ограничение в зависимости от гаджета, на котором игра, собственно, запускается.

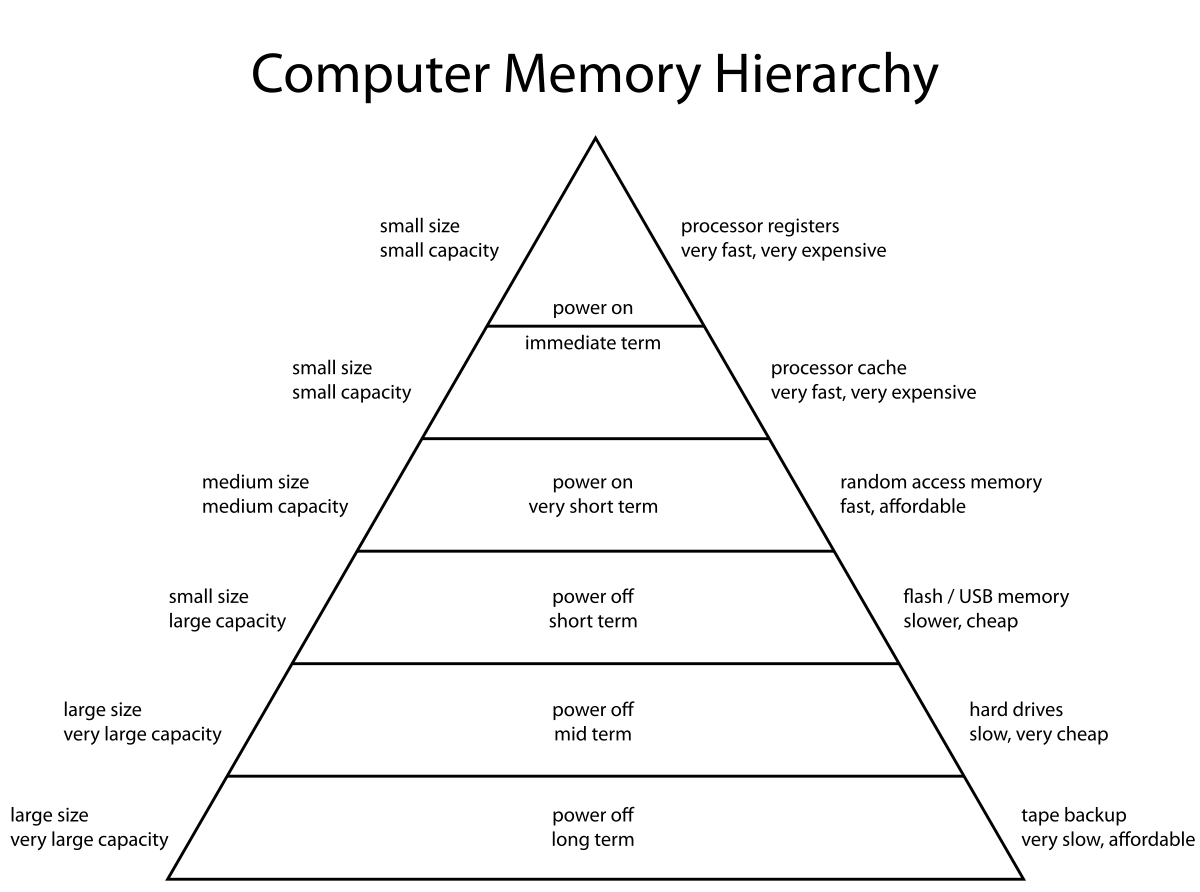
### 2. Фрагментация

Неприятный эффект, который появляется в процессе множественных аллокаций кусочков памяти различного размера. В результате вы получаете раздробленное на множество отдельных частей адресное пространство. Объединить эти части в единые блоки бОльшего размера не выйдет, так как часть памяти занята, и свободно перемещать ее мы не можем.

Фрагментация на примере последовательных аллокаций и освобождений блоков памяти

В итоге: имеем может и достаточно свободной памяти количественно, но не качественно. И очередной запрос, скажем, "выделить место под аудио дорожку" аллокатор удовлетворить не сможет, т.к единого куска памяти такого размера просто нет.

### 3. Кэш процессора

 Иерархия памяти компьютера

Кэш современного процессора - это некоторое промежуточное звено, которое связывает главную память (ОЗУ) и непосредственно регистры процессора. Так получилось, что доступ к памяти на чтение/запись - это весьма медленные операции (если говорить о кол-ве тактов CPU, необходимых для выполнения). Поэтому существует некоторая иерархия кэша (L1, L2, L3, и тд), которая позволяет как бы "по некоторому предсказанию" подгружать данные из ОЗУ, или неспешно их вытеснять в более медленную память.

Размещение однотипных объектов подряд в памяти позволяет "значительно" ускорить процесс их обработки (если обработка происходит последовательно), так как в этом случае проще предсказать, какие данные понадобятся дальше. И под "значительно" понимается прирост производительности в разы. Об этом неоднократно говорили разработчики движка Unity в своих [докладах на GDC](https://www.youtube.com/watch?v=EWVU6cFdmr0).

### 4. Multi-Threading

**Обеспечение безопасного доступа к разделяемой памяти в многопоточной среде** является одной из основных проблем, которую вам придется решать при создании собственного игрового движка/игры/любого другого приложения, которое использует несколько потоков для достижения бОльшей производительности. Современные вычислители устроены весьма нетривиальным образом. У нас имеется и сложная структура кэша, и несколько ядер вычислителя. Все это при неправильном использовании может привести к ситуациям, когда разделяемые данные вашего процесса будут повреждены в результате работы нескольких потоков (если они одновременно попытаются работать с этими данными без контроля доступа). В самом простом случае, это будет выглядеть следующим образом: Я не хочу углубляться в тему многопоточного программирования, так как множество его аспектов выходит очень сильно за рамки изложения статьи или даже целой книги.

### 5. Malloc/free

Операции по выделению/освобождению не происходят мгновенно. На современных ОС, если мы говорим про Windows/Linux/MacOS они реализованы хорошо и работают *в большинстве ситуаций быстро*. Но потенциально это весьма трудоемкие операции. Мало того, что это является системным вызовом, так еще в зависимости от реализации это может потребовать время для поиска подходящего куска памяти (First Fit, Best fit, и тд.) или поиска места для вставки и/или мерджа освобожденного участка.

Кроме того, свеже-выделенная память может быть в действительности не отображена на реальные физические страницы, что при первом обращении также может потребовать некоторое время.

Это детали реализации, а что с применимостью? Malloc/new не имеют представления о том, в каком месте, как и для чего вы их вызвали. Они выделяют память (в худшем случае) размером 1 KiB и 100 MiB одинаково... одинаково плохо. Непосредственно стратегия использования отдается на откуп либо программисту, либо тому, кто реализовал среду исполнения вашей программы.

### 6. Memory corruption

[Как говорит вики](https://en.wikipedia.org/wiki/Memory_corruption), это одна из самых непредсказуемых ошибок, проявляется только в ходе работы программы, и чаще всего вызвана непосредственно ошибками в написании этой программы. Но что эта проблема из себя представляет? К счастью (или сожалению), она не связана с коррумпированностью вашего компьютера. Скорее она отображает ситуацию, когда вы пытаетесь работать с памятью, которая вам *не принадлежит*. Сейчас поясню:

1. Это может быть попытка чтения/записи в участок не выделенной памяти
2. Выход за границы блока памяти, предоставленного вам. Это проблема как бы частный случай проблемы (1), но она хуже тем, что система скажет вам о том, что вы вышли за границы только тогда, когда вы покинете пределы отображенной для вас страницы. Т.е потенциально, эту проблему очень трудно отловить, т.к ОС в состоянии реагировать, только если вы покинете пределы отображенных вам виртуальных страниц. Вы можете попортить память процесса и получить очень странную ошибку из того места, откуда ее совсем не ждали.
3. Освобождение уже освобожденного (звучит странно) или еще не выделенного участка памяти
4. и т.д.

В С/С++, где есть арифметика указателей, с этим вы столкнетесь на раз-два. Однако в Java Runtime придется изрядно попотеть, чтобы получить подобного рода ошибку (сам не пробовал, но думаю, что такое возможно, иначе жизнь была бы слишком простой).

### 7. Утечки памяти

Является частным случаем более общей проблемы, встречающейся во многих языках программирования. Стандартная библиотека C/C++ предоставляет доступ к ресурсам ОС. Это могут быть файлы, сокеты, память и т.д. После использования ресурс должен быть корректно закрыт и занятую им память следует освободить. И если говорить конкретно об освобождении памяти - накапливающиеся утечки в результате работы программы могут привести к "out of memory" ошибке, когда ОС будет не способна удовлетворить очередной запрос на аллокацию. Часто разработчик просто забывает освободить использованную память по тем или иным причинам.

Здесь же стоит добавить про корректное закрытие и освобождение ресурсов на GPU, т.к ранние драйверы не позволяли возобновить работу с видеокартой если предыдущий сеанс был завершен неверно. Только перезагрузка системы могла решить эту проблему, что является весьма сомнительным - заставлять юзера перезагружать систему после работы вашего приложения.

### 8. Dangling pointer

[**Висячий указатель**](https://en.wikipedia.org/wiki/Dangling_pointer) - некоторый жаргон, описывает ситуацию, когда указатель ссылается на недопустимое значение. Подобная ситуация может запросто возникнуть при использовании классических C-style указателей в C/C++ программе. Предположим, вы выделили память, сохранили адрес на нее в указателе p, а потом освободили память (см. пример кода):

// Выделяем память  
void\* p = malloc(size);  
// ... Делаем что-то с полученной памятью  
// Благополучно освобождаем  
free(p);  
// Куда теперь указывает p?  
// \*p == ?

Указатель хранит некоторое значение, которое мы можем интерпретировать как адрес блока памяти. Так получилось, что мы не можем утверждать, является ли этот блок памяти валидным, или нет. Только программист, основываясь на тех или иных соглашениях, может оперировать указателем. Начиная с C++11 в стандартную библиотеку были введены ряд дополнительных указателей "smart pointers", которые позволяют в некотором плане ослабить контроль ресурсов со стороны программиста за счет использования дополнительной мета-информации внутри себя (об этом позже).

Как частичное решение можно использовать *специальное значение* указателя, которое будет сигнализировать нам о том, что по этому адресу ничего нет. В C в кастве этого значения используется макрос NULL, а в C++ ключевое слово языка nullptr. Решение это частичное, так как:

1. Значение указателя необходимо устанавливать вручную, поэтому программист может просто забыть это сделать.
2. nullptr или просто 0x0 входит в множество значений, принимаемых указателем, что не есть хорошо, когда особое состояние объекта выражается через его обычное состояние. Это некоторе legacy, и по договоренности ОС не выделит вам участок памяти, адрес которого начинается с 0x0.

Пример кода с null:

// Делаем что-то с p  
free(p);  
p = nullptr;  
// Теперь значение p == nullptr и мы знаем, что он ни на что не ссылается

Можно в некоторой степени автоматизировать этот процесс:

void \_free(void\* &p) {  
free(p);  
p = nullptr;  
}  
  
// Делаем что-то с p  
\_free(p);  
// Теперь значение p == nullptr, и нам не надо   
// вручную устанавливать его

### 9. Тип памяти

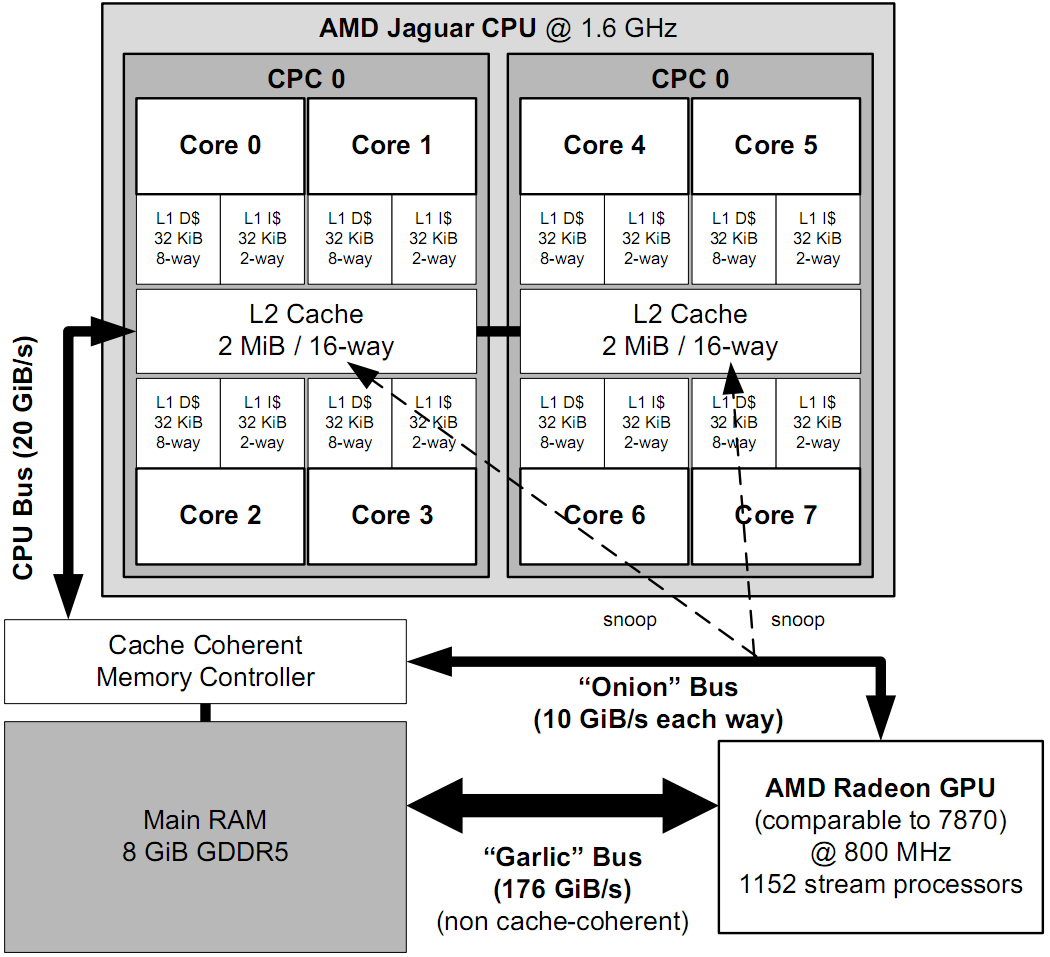
**RAM** - обычная оперативная память общего назначения, доступ к которой через центральную шину имеют все ядра вашего процессора и устройства периферии. Ее объем варьируется, но чаще всего речь идет о N гигабайтах, где N равняется 1,2,4,8,16 и тд. Вызовы malloc/free стремятся разместить желаемый вами блок памяти как раз в RAM компьютера.

**VRAM** (video memory) - видео память, поставляется вместе с видеокартой/видео-ускорителем вашего ПК. Она, как правило, меньшего объема чем RAM (порядка 1,2,4 GiB), однако обладает большим быстродействием. Распределением этого типа памяти занимается драйвер видеокарты, и чаще всего прямого доступа к ней вы не имеет.

На PlayStation 4 такого разделения нет, и вся оперативная память представлена едиными 8 гигабайтами на GDDR5. Поэтому все данные, как для процессора, так и для видео-ускорителя лежат рядом.

Хороший менеджмент ресурсов в игровом движке включает в себя грамотное распределение памяти как в Main RAM, так и на стороне VRAM. Здесь можно столкнуться с **дублированием**, когда одни и те же данные будут находятся там и там, или с **избыточным трансфером** данных с RAM на VRAM и обратно.

**В качестве иллюстрации ко всем озвученным проблемам**: можно посмотреть на аспекты устройства компьютеров на примере архитектуры PlayStation 4 (рис.). Здесь представлен центральный процессор, 8 ядер, кэши уровня L1 и L2, шины данных, оперативная память, графический ускоритель и т.д. С полным и подробным описанием можно ознакомиться в книге Джейсона Грегори ["Game Engine Architecture"](https://www.gameenginebook.com).

 Архитектура PlayStation 4

# Общие подходы к решению

Универсального решения нет. Но есть набор некоторых моментов, на которых следует акцентировать внимание, если вы собираетесь реализовывать мануально аллокацию и управление памятью в вашем приложении. Сюда можно отнести и контейнеры, и специализированные аллокаторы, стратегии выделения памяти, дизайн системы/игры, менеджеры ресурсов и другое.

## Типы аллокаторов

Использование специальных аллокаторов памяти основывается на следующей идее: вы знаете, какого размера, в какой моменты работы и в каком месте вам понадобятся участки памяти. Следовательно, вы можете выделить необходимую память, как-то ее структурировать и использовать / переиспользовать. В этом состоит общая идея/концепция использования специальных аллокаторов. Какие они бывают (разумеется не все) можно посмотреть далее:

1. **Linear allocator** Представляет буфер непрерывного адресного пространства. В ходе работы позволяет выделять участки памяти произвольного размера (такие, что они умещаются в буфер). Но освободить всю выделенную память можно только за 1 раз. Т.е произвольный участок памяти не может быть освобожден - он будет оставаться как бы *занятым* до тех пор, пока весь буфер не будет помечен как чистый. Такая конструкция обеспечивает выделение и освобождение за O(1), что дает гарантию скорости при любых условиях. Типичный use-case: в процессе обновления состояния процесса (каждый кадр в игре) вы можете использовать LinearAllocator для выделения tmp буферов для каких-либо технических нужд: обработка ввода, работа со строками, парсинг команд ConsoleManager в режиме отладки и т.д.
2. **Stack allocator** Модификация линейного аллокатора. Позволяет освобождать память в порядке, обратном порядку выделения, иными словами, ведет себя как обычный стек по принципу LIFO. Может быть весьма полезен для проведения нагруженных математических вычислений (иерархия трансформаций), для реализации работы скриптовой подсистемы, для любых вычислений, где наперед известен указанный порядок освобождения памяти. Простота конструкции обеспечивает O(1) скорость выделения и освобождения памяти.
3. **Pool allocator** Позволяет выделять блоки памяти одинакового размера. Может быть реализован как буфер непрерывного адресного пространства, разбитый на блоки заранее установленного размера. Эти блоки могут образовывать связный список. И мы всегда знаем, какой блок отдать при аллокации следующим. Эту мета информацию можно сохранить в самих блоках, что накладывает ограничение на минимальный размер блока (sizeof(void\*)). В реальности это не критично. Поскольку все блоки имеют единый размер, нам не принципиально, какой блок возвращать, а следовательно все операции по выделению/освобождению могут быть выполнены за O(1).
4. **Frame allocator** Линейный аллокатор но только с привязкой к текущему кадру - позволяет делать tmp выделения памяти и затем при смене кадра автоматически все освобождать. Его стоит выделить отдельно, так как это некоторая глобальная и уникальная сущность в рамках runtime игры, а поэтому его можно сделать весьма внушительного размера, скажем пара десятков MiB, что будет весьма кстати при загрузке ресурсов и их обработке.
5. **Double frame allocator** Представляет собой двойной frame allocator, но с некоторой особенностью. Он позволяет выделять память в ткущем кадре, и использовать ее как в текущем, так и в следующем кадрах. Т.е память, которую вы выделили в кадре N, будет освобождена только после N+1 кадра. Реализуется это переключением активного frame для выделения в конце каждого кадра. Но этот тип аллокатора как и предыдущий накладывает ряд ограничений на время жизни объектов, созданных в выделенной им памяти. Поэтому вы должны отдавать себе отчет о том, что по завершении кадра данные просто станут невалидными, и повторное к ним обращение может повлечь за собой серьезные проблемы.
6. **Static allocator** Аллокатор такого типа распределяет память из буфера, полученного, например, на этапе эта запуска программы, либо захваченного на стеке в фрейме функции. По типу это может быть совершенно любой аллокатор: linear, pool, stack. Почему он называется *статический*? Размер захватываемого буфера памяти должен быть известен **на этапе компиляции** программы. Это накладывает существенное ограничение: объем доступной этому аллокатору памяти не можем изменяться во время работы. Но каковы преимущества? Используемый буфер будет автоматически захвачен и потом освобожден (либо по завершении работы, либо при выходе из функции). Это не нагружает кучу, избавляет вас от фрагментации, позволяет быстро выделить память по месту. Можно посмотреть на примере кода использование этого аллокатора, если вам необходимо разбить строку на подстроки и что-то с ними сделать: Также можно отметить, что использование памяти из стека в теории значительно эффективнее, т.к. стек фрейм текущей функции с большой вероятностью уже будет в кэше процессора.

Все эти аллокаторы так или иначе решают проблемы с фрагментацией, с нехваткой памяти, со скоростью получения и освобождения блоков необходимого размера, с временем жизни объектов и занимаемой ими памятью.

Следует так же отметить, что правильный подход к дизайну интерфейсов позволит вам создавать своего рода *иерархии* аллокаторов, когда, например: pool выделит память из frame alloc, а frame alloc в свою очередь выделит память из linear alloc. Подобную структуру можно продолжить и дальше, адаптируя под свои задачи и нужды.

Подобный интерфейс для создания иерархий мне видится следующим образом:

class IAllocator {  
public:  
virtual void\* alloc(size\_t size) = 0;  
virtual void\* alloc(size\_t size, size\_t alignment) = 0;  
virtual void free (void\* &p) = 0;  
}

В него вы можете обернуть стандартные для вышей платформы malloc/free операций, что также даст дополнительную гибкость. Более того, подобные классы дают вам больше возможностей для профилирования, вы с легкостью сможете добавить подсчет необходимой статистики внутри реализации этих методов. В этот интерфейс можно было бы добавить синтаксический сахар по созданию/удалению объектов с использованием шаблонов, но лучше оставить это для следующего раза.

## Умные указатели

Smart pointer - это некоторая разновидность указателей в стандартной библиотеке C++ начиная с С++11 (не говорите про boost, разговор только о стандарте). Представляет собой класс-обертку, который хранит собственно сам указатель, осуществляет операции над ним и содержит дополнительную мета-информацию о том, как и когда освободить память и удалить объект. В этом месте мы немного отходим от непосредственно выделения памяти к скорее отслеживанию времени жизни объекта.

Так что же позволяют делать умные указатели? В самом простом случае это: 1. Автоматическое удаление объекта и освобождение памяти 2. Контроль доступа (уникальность/раздельность) 3. Больший контроль типа

Следующие виды умных указателей являются основными и самыми распространенными:

1. **Unique pointer** Позволяет иметь в программе только 1 указатель на объект (уникальность доступа). Как только unique pointer удаляется, сразу же удаляется используемый объект и освобождается занятая им память. Хорошо подходит для создания указателей на файловые дескрипторы, т.к. чаще всего только 1 участник может читать/писать файл. Если вы передаете указатель на объект из uniquePtr1 в uniquePtr2, то значение в uniquePtr1 инвалидируется, т.к допускается только 1 владелец объекта.
2. **Shared pointer** Указатели раздельного доступа с автоматическим подсчетом ссылок (reference counting). Позволяют нескольким участникам ссылаться на один объект, не беспокоясь о том, кто в итоге должен удалить его. Поскольку подсчет ссылок автоматический, то, как только последний ссылающийся указатель удален, объект также будет удален. Однако такой вид указателя не лишен недостатков. Во-первых, он не способен бороться с циклическими ссылками, когда объекты ссылаются друг на друга. В таком случае вы гарантированно получите утечку памяти. Во-вторых, в много-поточной среде работа с указателями раздельного доступа накладывает свои ограничения на консистентность памяти и раздельность доступа потоками.
3. **Weak pointer** Слабая ссылка на объект. Позволяет хранить указатель на объект раздельного доступа, но не удерживать его. Что это значит? Вы может создать объект и сохранить указатель на него в shared pointer. До тех пор, пока последний shared pointer не удален, объект будет оставаться в памяти. Однако, вы можете создать из shared pointer weak pointer. Таким образом, если сильные (shared) указатели остались на объект, то из weak pointer вы можете получить shared pointer. А если нет - то weak pointer инвалидируется, и вы узнаете о том, что объект, на который вы хранили указатель уже удален. Недостатком как shared, так и weak pointer является необходимость хранить дополнительно meta-data для каждого объекта. Здесь храниться и кол-во ссылок на объект, способ его удаления и т.д. В общем случае, это O(N) overhead по памяти, где N - кол-во создаваемых объектов. Однако такую жертву можно считать допустимой, т.к в проекте на тысячи строк кода вам будет тяжело договориться с другим программистом о том, кто ответственен за удаление того или иного объекта. В противном случае висячие ссылки и утечки памяти гарантированы.

Финальная мысль в этой части: использовать умные указатели необходимо с также некоторой долей осторожности. Например, используя shared pointer, вы неявно соглашаетесь с тем, что ваш объект (на который вы ссылаетесь через этот указатель) будет *где-то когда-то кем-то* удален. И это может быть не самый удачный момент работы вашей программы. Кроме это вы должны отдавать себе отчет о расходуемой памяти на meta-info и о том, как данные, на которые вы ссылаетесь через умный указатель будут попадать в кэш. Пример:

/\* Как не стоит делать \*/  
/\* Создадим массив объектов, используя shared pointer \*/  
Array<TSharedPtr<Object>> objects;  
objects.add(newShared<Object>(...));  
...  
objects.add(newShared<Object>(...));

/\* Как можно было бы сделать (не тратимся на meta-info и попадаем в кэш) \*/  
Array<Object> objects;  
objects.emplace(...);  
...  
objects.emplace(...);

Пример достаточно вымученный. Однако во многих ситуация мы можем отказаться от использования достаточно тяжеловесных умных указателей. Об этом далее.

## Unique id

Никто не заставляет нас использовать указатели, чтобы обращаться к объектам. Мы можем использовать идентефикаторы (id/identificator), выраженные целым числом, строкой, или еще как-то. Каковы преимущества данного подхода:

1. **Контроль доступа** Мы храним не сам объект, а его id. Чтобы обратиться к нему, нам необходим **посредник или сервис**, который знает, где найти и как обработать этот объект по его id.
2. **Контроль времени жизни** Даже если мы забудем удалить объект, то ответственный за него сервис сделает это за нас (в крайнем случае, в конце работы этого сервиса)
3. **Слабые ссылки и разделение доступа** Если объект с id был удален, то сервис скажет нам об этом, когда мы обратимся к нему по этому id.
4. **Расход памяти и порядок размещения** Сервис сам будет в состоянии разместить объекты в памяти более эффективным способом в рамках решения поставленной задачи. Мы, как держатели id, даже не узнаем об этом.

Очевидным недостатками такого подхода являются: необходимость контролировать уникальность id, ограничивать количество объектов, которые могут быть адресованы через id, жертвовать контролем и использовать посредника для доступа к объекту.

Подобные id повсеместно используют в современных приложения, библиотеках (Vulkan, OpenGL), игровых движках (Godot, CryEngine). Об EntityID можно почитать [в документации к упомянутому уже CryEngine](https://docs.cryengine.com/display/CEPROG/Entity).

Рассмотрим простой пример, когда id представлен двумя числами: индекс и поколение. Индекс говорит нам о том, где конкретно лежит объект (в какой ячейке массива), а поколение указывает на но, был ли объект удален или нет.

/\* Пример структуры идентефикаторы \*/  
class ID {  
uint32 index;  
uint32 generation;  
}

/\* Пример класса-обработчика / менеджера \*/  
class ObjectManager {  
public:  
ID create(...);  
void destroy(ID);  
void update(ID id, ...);  
private:  
Array<uint32> generations;  
Array<Objects> objects;  
}

Теперь по ID обработчик будет в состоянии найти объект или определить, что объекта с таким ID нет. Это можно сделать следующим образом:

generation = generations[id.index];  
if (generation == id.generation)   
then /\* нашли такой объект \*/  
else /\* не нашли, объект был уже удален \*/

При удалении объекта по его id мы просто должны увеличить счетчик generation на 1 у соответствующего id из массива ids.

## Контейнеры

В стандартной библиотеке языка C++ есть множество классов, которые облегчают работу с тем или иным видом объектов. И здесь это не реклама std, вы можете самостоятельно реализовать **единожды** необходимые вам контейнеры, а затем **переиспользовать** их неограниченное число раз. В первую очередь обратите внимание на следующие классы:

* Linked list - связный список
* Array - динамический/статический массив
* Queue - очередь
* Stack - стек
* Map - ассоциативный контейнер
* Set - множество

Почему я их упоминаю в статье про менеджмент памяти? Потому что они контролируют доступ к объектам внутри себя и исключают возможность memory corruption. Вызывая необходимые методы вы с легкостью сможете добавлять/удалять объекты, получать на них ссылки, итерироваться по ним, не задумываясь о том, где и как они лежат.

# Общие идеи

Здесь мне бы хотелось уйти от конкретики к более абстрактным вещам, которые касаются не столько кода, сколько архитектуры и дизайна системы в целом. Ряд наблюдений, мыслей, которые мне удавалось увидеть/прочесть в том или ином источнике литературы за долгое время.

## Под конкретные задачи

При создании системы, разработчик как никто иной должен знать, какие данные и как он будет обрабатывать. На основе этого он сможет выбрать ту стратегию выделения памяти и работы с ней, которая наилучшим (в рамках задачи) образом подходит ему. Как было сказано ранее, библиотечные функции malloc/free не знают, как и для чего вы их вызвали, поэтому полагаться на их производительность при решении весьма специфических задач не стоит.

Что требуется брать в расчет? Вы можете определить, сколько памяти вам потребуется (нижняя/верхняя граница), в какие моменты работы программы, какие задачи потребуют множество аллокаций, а какие нет. Можно порассуждать о том, можно ли часть аллокаций убрать, разместить данные на стеке, перенести запросы на выделение из нагруженных частей программы в менее нагруженные.

СryEngine Sandbox: как пример среды для разработки игр

Крупные игровые движки, такие как Unreal, Unity, CryEngine и т.д, ничего не знают о том, какую игру вы делаете. Да, они могут быть заточены под определенные механики, жанры, но в общем случае - только вы сможете настроить систему таким образом, что она будет в состоянии удовлетворить ваши запросы на размещение тех или иных ресурсов в памяти компьютера.

## Pre-allocating

Заране получить всю необходимую память для работы программы, разместить в ней необходимые объекты и/или распределять эту память под свои нужды уже в процессе работы.

Преимущество такого подхода очевидно: никаких запросов malloc/free на выделение памяти в течение работы. Отсюда вытекает и тот факт, что ОС не скажет вам "run out of memory", т.к все необходимое вы получили заранее и не более. Теперь вы сможете работать с памятью в том стиле, который требуется для решения ваших задач (выравнивание, попадание в кэш, упаковка данных подряд и т.д).

Но такой подход очень сложен в реализации. Он требует ручного распределения памяти среди всех объектов вашей системы. Более того, вы неизбежно можете столкнуться с перерасходом и большим кол-вом неиспользуемой памяти. Если ваша система будет делать множество динамических аллокаций в процессе работы, то такой подход неизбежно приведет ко всем тем проблемам с malloc/free, которые были озвучены ранее: фрагментация, долгие вызовы, нехватка памяти.

## Не надо бояться динамической памяти

В наше время многие системы требуют динамического выделения памяти в процессе работы. И это нормально: вы не можете знать заранее, какого размера вам на вход попал файл, текстура, меш и т.д. Современные вычислители смогут позволить себе пару тысяч динамических аллокаций без потерь в производительности.

Однако не стоит увлекаться: если вы знаете, как и какая подсистема использует ресурсы, то лучше двигаться в сторону специализированных аллокаторов, адаптированных под конкретные цели. Большинство open-source движков, исходный код которых мне удавалось посмотреть, по такому принципу и работают. Где они могут, там оптимизируют, где нет - отдают все на откуп malloc/free.

## Дизайн из ограничений

На GDC компания CD Project Red [представила интересный доклад](https://www.youtube.com/watch?v=9vEfH9SJ9mY), в котором дизайнер уровней рассказал, как они создавали архитектуру игровых локаций "The Witcher: Bloode and Wine" в условиях четких (количественных) ограничений на расход памяти и время подгрузки локаций. Им приходилось выверять количество геометрии, которая будет загружена в памяти, выстраивать здания таким образом, что движок будет способен подгрузить все необходимые данные налету, пока игрок идет из точки А в точку Б. 

[В одной из статей](https://www.gnomon.edu/blog/discover-naughty-dog-s-environment-art-workflow-for-uncharted-4) дизайнер локаций из Naughty Dog также упомянул, что им приходилось при разработке "Uncharted 4: A Thief's End" оптимизировать мир игры таким образом, что в любой момент времени вся обозримая часть (то, что находиться в кадре) локации могла быть успешно обработана движком игры. 

# Заключение

В конце хотелось бы отметить, что управление памятью программы, как и управление други ресурсами, выходит далеко за рамки простых менеджеров памяти и структур данных. В рамках игровых движков существует множество аспектов, которые так или иначе связаны с темой этой статьи. Это может быть и загрузка/хранение ресурсов, отслеживание времени жизни игровых объектов, обеспечение безопасности много-поточного доступа и т.д. Со всеми моментами можно и необходимо познакомиться, прочитав специальную литературу по каждой из тем (конечно, если вам это интересно).

# Литература и полезные ссылки

* Хотелось бы еще раз упомянуть книгу Джейсона Грегори ["Game Engine Architecture"](https://www.gameenginebook.com). В ней он рассматривает многие аспекты создания игр и игровых движков, включая звук, графику, физику, математику и т.д. Часть моментов, касающихся работы с памятью, также детально освещается в этой книге.
* [Custom memory allocators](https://github.com/mtrebi/memory-allocators) - здесь вы может ознакомиться с деталями реализации аллокаторов, посмотреть исходный код на C++ и замеры. Это отличный пример для тех, кто желает погрузиться в рутину реализации собственных менеджеров памяти.
* [Smart pointers](https://en.cppreference.com/book/intro/smart_pointers) - можно ознакомиться с умными указателями, деталями использования и полным набором функциональности.
* [Start Pre-allocating And Stop Worrying](http://gamesfromwithin.com/start-pre-allocating-and-stop-worrying) - еще пара размышлений на тему менеджмента памяти