Пример эффективного аллокатора памяти с фиксированными блоками на C++

Аллокатор памяти с фиксированными блоками на C++, который увеличивает производительность системы и обеспечивает защиту от фрагментации кучи.

CMake используется для создания файлов сборки. CMake — это бесплатное программное обеспечение с открытым исходным кодом. Поддерживаются Windows, Linux.

Описание

Пользовательские аллокаторы памяти с фиксированными блоками используются для решения как минимум двух типов проблем, связанных с памятью.

Во-первых, глобальные выделения и удаления из кучи могут быть медленными и недетерминированными. Вы никогда не знаете, сколько времени займет работа менеджера памяти.
Во-вторых, исключение возможности ошибки выделения памяти из-за фрагментированной кучи - обоснованное опасение, особенно в критически важных системах.
Даже если система не считается критически важной, некоторые встраиваемые системы рассчитаны на работу в течение нескольких недель или лет без перезагрузки. В зависимости от шаблонов распределения и реализации кучи, длительное использование кучи может привести к сбоям в работе кучи.
Типичным решением является статическое объявление всех объектов и полный отказ от динамического выделения. Однако статическое выделение может привести к нерациональному использованию памяти, поскольку объекты, даже если они активно не используются, существуют и занимают место. Кроме того, реализация системы на основе динамического выделения может предложить более естественную архитектуру, чем статическое объявление всех объектов.
Распределители памяти с фиксированными блоками - идея не новая, уже давно разрабатываются различные виды пользовательских распределителей памяти.

Здесь представляется простая реализацию аллокатора на C++, которую успешно можно использовать в различных проектах.
Представленное здесь решение будет:

- Быстрее, чем глобальная куча
- Устранять ошибки фрагментации памяти кучи
- Не требует дополнительных накладных расходов на хранение (за исключением нескольких байт статической памяти)
- Быть простым в использовании
- Занимать минимум места в коде

Рассмотрим простой класс, выделяющий и возвращающий память, который обеспечит все вышеперечисленные преимущества.

Рециркуляция памяти

Основная схема управления памятью заключается в повторном использовании памяти, полученной при выделении объектов. После создания хранилища для объекта оно никогда не возвращается в кучу. Вместо этого память утилизируется, позволяя другому объекту того же типа повторно использовать это место. Здесь реализован класс под названием Allocator, который выражает эту технику.
Когда приложение удаляет объект с помощью Allocator, блок памяти для одного объекта освобождается для повторного использования, но на самом деле не передается обратно в менеджер памяти.
Освобожденные блоки сохраняются в связанном списке, называемом free-list, чтобы быть снова выделенными для другого объекта того же типа.
При каждом запросе на выделение Allocator сначала проверяет free-list на наличие существующего блока памяти.
Только если таковых нет, создается новый.
В зависимости от желаемого поведения Allocator, память берется либо из глобальной кучи, либо из статического пула памяти с одним из трех режимов работы:

1. Блоки из кучи
2. Пул из кучи
3. Статический пул

Куча против пула

Класс Allocator способен создавать новые блоки из кучи или пула памяти, когда свободный список не может предоставить существующий блок. Если используется пул, вы должны заранее указать количество объектов. На основе общего количества объектов создается пул, достаточно большой для обработки максимального количества экземпляров. Получение блочной памяти из кучи, не имеет ограничений по количеству - создавайте столько новых объектов, сколько позволяет память.

Режим *блоков из кучи* выделяет из глобальной кучи новый блок памяти для одного объекта по мере необходимости. Деаллокация помещает блок в свободный список для последующего повторного использования.

В случае, когда свободный список пуст, создание новых блоков из кучи избавляет от необходимости устанавливать ограничение на количество объектов. Этот подход обеспечивает динамическую работу, поскольку количество блоков может расширяться во время выполнения.
Недостатком является потеря детерминированности выполнения во время создания блоков.

Режим *пула из кучи* создает единый пул из глобальной кучи для хранения всех блоков. Пул создается с помощью оператора **new** при построении объекта Allocator. Затем Allocator предоставляет блоки памяти из пула во время выделений.

В режиме *статического пула* для хранения всех блоков используется один пул памяти, обычно расположенный в статической памяти. Статический пул памяти не создается Allocator, а предоставляется пользователем класса.

Режимы *пула из кучи* и статического пула обеспечивают постоянное время выполнения выделения, поскольку менеджер памяти никогда не участвует в получении отдельных блоков. Это делает новую операцию очень быстрой и детерминированной.

Описание классов

Интерфейс класса очень прост
Allocate() возвращает указатель на блок памяти,
Deallocate() освобождает блок памяти для повторного использования.
Конструктор отвечает за установку размера объекта и, при необходимости, за выделение пула памяти.

Аргументы, передаваемые в конструктор класса, определяют, откуда будут браться новые блоки.

Аргумент size управляет фиксированным размером блока памяти.
Аргумент objects задает, сколько блоков разрешено. Значение 0 означает получение новых блоков из кучи по мере необходимости, в то время как любое другое ненулевое значение означает использование пула, кучи или статического, для обработки указанного количества экземпляров.

Аргумент memory - это указатель на необязательную статическую память. Если память равна 0, а объекты не равны 0, Allocator создаст пул из кучи.

Пул статической памяти должен иметь размер x объектов в байтах.
Аргумент name опционально задает имя аллокатора, что полезно для сбора метрик использования аллокатора.

```
class Allocator
{
public:
    Allocator(size_t size, UINT objects=0, CHAR* memory=NULL, const CHAR* name=NULL);
```

…
В примерах ниже показано, как каждый из трех режимов работы управляется через аргументы конструктора.

```
// Режим блоков кучи с неограниченными блоками по 100 байт
Allocator allocatorHeapBlocks(100);

// Режим пула кучи с 20, 100 байтовыми блоками
Allocator allocatorHeapPool(100, 20);

// Режим статического пула с 20, 100 байтовыми блоками
char staticMemoryPool[100 * 20];

Allocator allocatorStaticPool(100, 20, staticMemoryPool);
```

Чтобы несколько упростить метод статического пула, используется простой шаблонный класс AllocatorPool<>.
Первый аргумент шаблона - тип блока, второй аргумент - количество блоков.

```
// Режим статического пула с 20 блоками размера MyClass
AllocatorPool<MyClass, 20> allocatorStaticPool2;
```

При вызове Allocate() возвращается указатель на блок памяти размером одного экземпляра.

При получении блока проверяется свободный список, чтобы определить, не существует ли уже этот блок. Если да, то просто удалите существующий блок из free-list'a и верните указатель на него; в противном случае создайте новый из пула/кучи.

```
void* memory1 = allocatorHeapBlocks.Allocate(100);
```

Deallocate() помещает адрес блока в стек. На самом деле стек реализован как односвязный список (free-list), но класс только добавляет/удаляет из головы списка, поэтому его поведение соответствует стеку. Использование стека делает выделение/удаление очень быстрым. Никакого поиска по списку - просто выделите или выгрузите блок и работайте.

```
allocatorHeapBlocks.Deallocate(memory1);
```

Есть возможность связывать блоки между собой в свободном списке, не занимая дополнительного места для хранения указателей.
Если, например, используется глобальный оператор new, то сначала выделяется память, а затем вызывается конструктор.

Процесс уничтожения обратный: вызывается деструктор, затем освобождается память. После выполнения деструктора, но до того, как память будет освобождена, она больше не используется объектом и освобождается для использования в других целях, например, для следующего указателя.

Поскольку классу Allocator необходимо хранить удаленные блоки, во время оператора delete мы помещаем следующий указатель списка в это неиспользуемое в данный момент пространство объекта.
Когда блок будет повторно использован приложением, указатель больше не понадобится и будет перезаписан вновь сформированным объектом. Таким образом, не возникает накладных расходов на хранение каждого экземпляра.
Использование освобожденного пространства объекта в качестве памяти для связывания блоков между собой означает, что объект должен быть достаточно большим, чтобы вместить указатель.
Код в списке инициализаторов конструктора гарантирует, что минимальный размер блока никогда не будет меньше размера указателя.
Деструктор класса освобождает место, выделенное во время выполнения, удаляя пул памяти или, если блоки были получены из кучи, обходя free-list и удаляя каждый блок.
Поскольку класс Allocator обычно используется как статический класс области видимости, он будет вызван только при завершении программы. В большинстве встраиваемых устройств приложение завершается, когда кто-то отключает питание от системы. Очевидно, что в этом случае деструктор не нужен.
Если вы используете метод блоков кучи, выделенные блоки не могут быть освобождены при завершении работы приложения, пока все экземпляры не будут занесены в свободный список. Поэтому все оставшиеся объекты должны быть «удалены» до завершения работы программы.
В противном случае вы получите отличную утечку памяти. В связи с этим возникает интересный момент. Разве Allocator не должен отслеживать как свободные, так и используемые блоки? В целом – нет. Если блок передается приложению через указатель, то приложение обязано вернуть этот указатель в Allocator с помощью вызова Deallocate() перед завершением программы. Таким образом, необходимо следить только за освобожденными блоками.

Использование кода
Для простоты использования были созданы макросы для автоматизации интерфейса в клиентском классе. Макросы предоставляют статический экземпляр Allocator и две функции-члена: оператор new и оператор delete. Перегружая операторы new и delete, Allocator перехватывает и обрабатывает все обязанности по выделению памяти для клиентского класса.

Макрос DECLARE_ALLOCATOR обеспечивает интерфейс заголовочного файла и должен быть включен в объявление класса следующим образом:

```
#include «Allocator.h»
```

```
class MyClass
{
    DECLARE_ALLOCATOR
    // остальное определение класса
};
```
Функция operator new вызывает Allocator, чтобы создать память для одного экземпляра класса. После выделения памяти, по определению, оператор new вызывает соответствующий конструктор класса. При перегрузке new можно взять на себя только обязанности по выделению памяти. Вызов конструктора гарантируется языком. Аналогично, при удалении объекта система сначала вызывает для нас деструктор, а затем выполняется оператор delete. Оператор delete использует функцию Deallocate() для сохранения блока памяти в свободном списке.
Известно, что при удалении класса, использующего базовый указатель, деструктор должен быть объявлен виртуальным. Это гарантирует, что при удалении класса будет вызван правильный деструктор производного класса.
Однако менее очевидно то, как виртуальный деструктор изменяет вызов перегруженного оператора delete класса.

Хотя оператор delete не объявлен явно, он является статической функцией, следовательно она не может быть объявлена виртуальной. Поэтому на первый взгляд можно предположить, что удаление объекта с базовым указателем не может быть направлено в нужный класс. В конце концов, вызов обычной статической функции с базовым указателем вызовет версию базового члена. Однако, как мы знаем, при вызове оператора delete сначала вызывается деструктор. При наличии виртуального деструктора вызов направляется в производный класс.
После выполнения деструктора класса вызывается оператор delete для этого производного класса. Таким образом, по сути, перегруженный оператор delete был направлен в производный класс через виртуальный деструктор. Поэтому, если выполняется удаление с использованием базового указателя, деструктор базового класса должен быть объявлен виртуальным. В противном случае будет вызван неправильный деструктор и перегруженный оператор delete.

Макрос IMPLEMENT_ALLOCATOR представляет собой часть интерфейса в исходном файле и должен быть помещен в область видимости файла.
IMPLEMENT_ALLOCATOR(MyClass, 0, 0)

После установки макросов вызывающая сторона может создавать и уничтожать экземпляры этого класса, а удаленные объекты будут утилизированы:

MyClass* myClass = new MyClass();
удалить myClass;

Для класса Allocator работает как одиночное, так и множественное наследование. Например, если предположить, что класс Derived наследует от класса Base, то следующий фрагмент кода будет законным.

```
Base* base = new Derived;
```
delete base;

Время выполнения

Во время выполнения Allocator изначально не будет иметь блоков в свободном списке, поэтому при первом вызове Allocate() будет получен блок из пула или кучи.
По мере выполнения системный спрос на объекты любого экземпляра Allocator будет колебаться, и новое хранилище будет выделяться только тогда, когда свободный список не сможет предложить существующий блок.

В конце концов, система установится на некотором пиковом количестве экземпляров, так что каждое выделение будет происходить из существующих блоков, а не из пула/кучи.
По сравнению с получением всех блоков с помощью менеджера памяти, класс экономит много вычислительной мощности.
При выделении указатель просто выдергивается из свободного списка, что делает его очень быстрым. При деаллокации указатель блока просто выталкивается обратно в список, что не менее быстро.

Необходимость использования аллокаторов

Прежде чем использовать аллокатор необходимо понять, - нужен ли вам аллокатор вообще. Если в вашем проекте нет требований к скорости выполнения или отказоустойчивости, то, вероятно, вам не нужен пользовательский аллокатор, и глобальная куча будет работать просто отлично.

С другой стороны, если вам требуется скорость и/или отказоустойчивость, аллокатор может помочь, а режим работы зависит от требований вашего проекта.

Архитектор критически важного проекта может запретить использование глобальной кучи. Однако динамическое распределение может привести к более эффективному или элегантному дизайну.

В этом случае можно использовать режим блоков кучи во время отладки ПО, чтобы получить метрики использования памяти, а затем для выпуска ПО переключиться на метод статических пулов, чтобы создать статически выделенные пулы, устранив таким образом все обращения к глобальной куче. Переключение между режимами осуществляется с помощью нескольких макросов времени компиляции.

В качестве альтернативы для приложения может подойти режим блоков кучи. Он использует кучу для получения новых блоков, но предотвращает ошибки фрагментации кучи и ускоряет выделение, когда свободный список заполнен достаточным количеством блоков. Это не реализовано в исходном коде из-за проблем с многопоточностью, выходящих за рамки этой статьи, можно просто заставить конструктор Allocator хранить статический список всех созданных экземпляров.

Запустите систему на некоторое время, затем в определенный момент пройдитесь по всем аллокаторам и выведите метрики, такие как количество блоков и имя, с помощью функций GetBlockCount() и GetName().
Метрики использования предоставляют информацию о размере фиксированного пула памяти для каждого аллокатора при переходе на пул памяти.
Всегда добавляйте на несколько блоков больше, чем максимальное измеренное количество блоков, чтобы обеспечить системе дополнительную устойчивость к нехватке памяти в пуле.

Отладка утечек памяти

Отладка утечек памяти может быть очень сложной, в основном потому, что куча - это «черный ящик», в котором не видны типы и размеры выделяемых объектов.
С помощью Allocator утечки памяти найти немного проще, поскольку Allocator отслеживает общее количество блоков. Повторный вывод (например, в консоль) GetBlockCount() и GetName() для каждого экземпляра аллокатора и сравнение различий должны выявить аллокатор с постоянно растущим количеством блоков.

Обработка ошибок

Ошибки выделения в C++ обычно отлавливаются с помощью функции new-handler. Если при попытке выделения памяти из кучи менеджер памяти дает сбой, пользовательская функция обработки ошибок вызывается через указатель функции new-handler. Присвоив адрес пользовательской функции new-handler, менеджер памяти может вызвать пользовательскую процедуру обработки ошибок. Чтобы сделать обработку ошибок в классе Allocator последовательной, выделения, превышающие объем памяти пула, также вызывают функцию, на которую указывает new-handler, централизуя все ошибки выделения памяти в одном месте.

```
static void out_of_memory()
{
    // функция new-handler вызывается аллокатором, когда в пуле заканчивается память
    assert(0);
}
int _tmain(int argc, _TCHAR* argv[])
{
    std::set_new_handler(out_of_memory);
...
```

Ограничения

Класс не поддерживает массивы объектов. Перегруженный оператор new[] создает проблему для метода утилизации объектов. При вызове этой функции аргумент size_t содержит общий объем памяти, необходимый для всех элементов массива. Создание отдельного хранилища для каждого элемента - не вариант, поскольку многократные вызовы new не гарантируют, что блоки находятся в смежном пространстве, что необходимо массиву. Поскольку Allocator работает только с блоками одинакового размера, массивы недопустимы.

Проблемы переноса

Allocator напрямую вызывает new_handler(), когда в статическом пуле заканчивается место для хранения, что может быть неприемлемо для некоторых систем.