Object-oriented dynamic memory reallocation

Yury A. Blinkov

Mechanics and Mathematics Department,
Saratov State University, Saratov, Russia
Department of Applied Probability and Informatics,
Peoples' Friendship University of Russia, Moscow, Russia

Mathematical modeling, november 10, 2021

Содержание

1. Введение
2. Алгоритмы «сборки мусора»
3. Алгоритмы и реализации для работы с «кучей»
4. ОО реализация динамического перераспределения памяти
5. Использование в GInv

Использование памяти

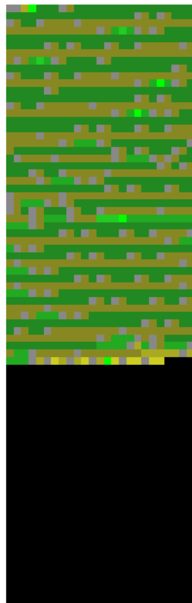
в современных языках программирования можно разделить на три категории по времени их «жизни» в программе:

1. все время «жизни», так называемая «статическая» память, а переменные, использующие эту память, часто называют статическими. Сюда также попадают и глобальные.
2. время выполнения функции, память выделяется в стеке, а переменные называют автоматическими, но в языках программирования используют слово «локальные».
3. время «жизни» определяет программист, если нет «сборки мусора» (garbage collection) **или** «сборка мусора», если она есть в выбранном языке программирования. Память выделяется в «куче».

Картинка целиком — это память для «кучи», выделенная для программы. Сначала она чёрная, то есть неиспользуемая, но постепенно начинается подсвечиваться ярко-жёлтым (операции записи) и ярко-зелёным (операции чтения). Со временем записанные фрагменты темнеют, чтобы показать развитие процесса во времени.

Можно заметить, что по ходу выполнения программа начинает игнорировать отдельные участки памяти. Они и считаются «мусором».

`https://github.com/kenfox/gc-viz`



«Сборка мусора»

была впервые применена Джоном Маккарти в 1959 году в среде программирования на разработанном им функциональном языке программирования Лисп.

Впоследствии она применялась в других системах программирования и языках, преимущественно — в функциональных и логических. Широко используемые в этих языках списки и основанные на них сложные структуры данных во время работы программ постоянно создаются, надстраиваются, расширяются, копируются, и правильно определить момент удаления того или иного объекта затруднительно.

Со второй половины 1980-х годов технология сборки мусора стала использоваться и в директивных (императивных), и в объектных языках программирования, а со второй половины 1990-х годов всё большее число создаваемых языков и сред, ориентированных на прикладное программирование, включают механизм сборки мусора либо как единственный, либо как один из доступных механизмов управления динамической памятью. В настоящее время она используется в Оберон, Java, Python, Ruby, C#, D, F#, Go и других языках.

По утверждению Вирта, разработчики языка Java за несколько лет до её создания «изучили исходные коды Оберона и, в частности, исходные коды оберононовских сборщиков мусора. Потом они испортили Оберон синтаксисом Си и назвали получившееся словом Java». Хотя от устного выступления нельзя требовать абсолютной точности формулировок, но во всяком случае несомненное сходство идеологий Оберона и Java (стремление к минимализму и строгой типизации, ограничение множественного наследования, автоматическое управление памятью) говорит о том, что здесь имеет место определённый консенсус относительно того, какие средства должны составлять ядро современного языка программирования общего назначения. Однако если в Обероне и его прямых наследниках минимализм остаётся во главе угла, разработчики Java пошли по пути экстенсивного наращивания возможностей языка.
Лекция Н. Вирта в Нижегородском государственном университете им. Н. И. Лобачевского

Java GC:

- G1
- Parallel
- Concurrent mark sweep collector (CMS)
- Serial
- C4 (Continuously Concurrent Compacting Collector)[31]
- Shenandoah
- ZGC

Язык GO:

- Использование утверждений (assertion) было сочтено ненужным.
- Переопределение методов и функций было исключено из соображений надёжности и эффективности компиляции: требование различного именования всех методов на одном уровне видимости устраняет необходимость сопоставлять списки параметров при компиляции вызовов функций и методов и исключает ошибочный вызов другого одноимённого метода.

C#

- Андерс Хейлсберг – датский инженер-программист, создатель Turbo Pascal, Delphi, C# и TypeScript.

D

- использует сборщик мусора для управления памятью, однако возможно и ручное управление с помощью перегрузки операторов new и delete, а также с помощью malloc и free, аналогично C. Сборщик мусора можно включать и выключать вручную, можно добавлять и удалять области памяти из его видимости, принудительно запускать частичный или полный процесс сборки. Существует подробное руководство, описывающее различные схемы управления памятью в D для тех случаев, когда стандартный сборщик мусора неприменим.

«Алгоритм пометок» (Mark and Sweep), заключается в следующем:

- для каждого объекта хранится бит, указывающий, достижим ли этот объект из программы или нет;
- изначально все объекты, кроме корневых, помечаются как недостижимые;
- рекурсивно просматриваются и помечаются как достижимые объекты, ещё не помеченные, и до которых можно добраться из корневых объектов по ссылкам;
- те объекты, у которых бит достижимости не был установлен, считаются недостижимыми.

Если два или более объектов ссылаются друг на друга, но ни на один из этих объектов нет ссылок извне, то вся группа считается недостижимой. Данный алгоритм позволяет гарантированно удалять группы объектов, использование которых прекратилось, но в которых имеются ссылки друг на друга. Такие группы часто называются «islands of isolation» (острова изоляции).

Алгоритм подсчёта ссылок

Другой вариант алгоритма обычный подсчёт ссылок.

- Его использование замедляет операции присваивания ссылок, но зато определение достижимых объектов тривиально — это все объекты, значение счётчика ссылок которых превышает нуль.
- Без дополнительных уточнений этот алгоритм, в отличие от предыдущего, не удаляет циклически замкнутые цепочки вышедших из употребления объектов, имеющих ссылки друг на друга.

GC это «кухня», «кухня» и еще раз «кухня»

- Сборка мусора потребляет вычислительные ресурсы для принятия решения о том, какую память освободить, даже если программист, возможно, уже знал эту информацию.
- Момент, когда мусор действительно будет собран, может быть непредсказуемым, что приводит к остановкам (паузы для сдвига / освобождения памяти), разбросанных по всему сеансу. Инкрементные, параллельные сборщики мусора и сборщики мусора в реальном времени решают эти проблемы с различными компромиссами.
- Самое главное: GC не учитывает «защищенный режим», «страничную организацию памяти» и самое «секретное оружие» современных компьютеров кэш (сверхоперативная память).

Алгоритмы и реализации для работы с «кучей»

- Задача выполнения запроса на выделение памяти состоит в обнаружении блока неиспользуемой памяти достаточного размера.
- Запросы к памяти удовлетворяются путем выделения частей из большого пула памяти, называемого кучей или свободным хранилищем.
- В любой момент времени некоторые части кучи используются, а некоторые «свободны» (не используются) и, следовательно, доступны для будущих распределений.
- Несколько проблем усложняют реализацию, например, внешняя фрагментация, которая возникает, когда есть много небольших промежутков между выделенными блоками памяти, что делает недействительным их использование для запроса выделения.
- Метаданные распределителя также могут увеличивать размер (по отдельности) небольших выделений.

Распределение блоков фиксированного размера

- При выделении блоков фиксированного размера, также называемом распределением пула памяти, используется список свободных блоков памяти фиксированного размера (часто все одного размера).
- Это хорошо работает для простых встроенных систем, где не требуется выделять большие объекты, но страдает от фрагментации, особенно с длинными адресами памяти. Однако из-за значительного сокращения накладных расходов этот метод может существенно улучшить производительность для объектов, которым требуется частое выделение / освобождение, и часто используется в видеоиграх.

Блоки друзей

- В этой системе память распределяется в несколько пулов памяти вместо одного, где каждый пул представляет собой блоки памяти с размером определенной степени двойки или блоки другой удобной прогрессии размера. Все блоки определенного размера хранятся в отсортированном связанном списке или дереве, и все новые блоки, которые формируются во время выделения, добавляются в соответствующие пулы памяти для последующего использования.
- Если запрашивается меньший размер, чем доступен, выбирается и разделяется наименьший доступный размер. Выбирается одна из полученных частей, и процесс повторяется до тех пор, пока запрос не будет завершен. Когда блок распределяется, распределитель начинает с самого маленького достаточно большого блока, чтобы избежать ненужного разбиения блоков.
- Когда блок освобождается, он сравнивается со своим приятелем. Если они оба свободны, они объединяются и помещаются в список блоков друзей большего размера.

Выделение Slab

- Этот механизм распределения памяти заранее выделяет блоки памяти, подходящие для размещения объектов определенного типа или размера.
- Эти фрагменты называются кешами, и распределителю нужно только отслеживать список свободных слотов кеша. Создание объекта будет использовать любой из свободных слотов кэша, а разрушение объекта добавит слот обратно в список свободных слотов кэша.
- Этот метод уменьшает фрагментацию памяти и является эффективным, поскольку нет необходимости искать подходящую часть памяти, поскольку достаточно любого открытого слота.

Реализации malloc/free

- 2020, компания Google представила новый вариант системы распределения памяти TCMalloc, которая используется во многих внутренних проектах Google. Код TCMalloc написан на C++ и распространяется под лицензией Apache. Для работы требуется наличие компилятора с поддержкой C++17 для языка C++, и C11 для языка Си (gcc 9.2+ или clang 9.0+). Из операционных систем поддерживается только Linux (x86, PPC).
  Примечательно, что с 2005 года существует ещё один вариант tcmalloc, который поставлялся в составе пакета gperftools (Google Performance Tools).

Реализации malloc/free

- 2021, mimalloc бесплатный компактный менеджер памяти общего назначения с открытым исходным кодом, разработанный Microsoft с акцентом на характеристики производительности. Библиотека составляет около 11000 строк кода и работает как замена malloc стандартной библиотеки C и не требует дополнительных изменений кода. mimalloc изначально был разработан для систем времени выполнения языков Lean и Koka. Исходный код лицензирован по лицензии MIT и доступен на GitHub.
- jemalloc реализация malloc общего назначения, для предотвращение фрагментации и поддержки масштабируемого параллелизма. jemalloc впервые начал использоваться в качестве распределителя libc FreeBSD в 2005 году.

International Symposium on Memory Management I

- 1992: http://www.informatik.uni-trier.de/~ley/db/conf/iwmm/iwmm92.html
- 1995: http://www.informatik.uni-trier.de/~ley/db/conf/iwmm/iwmm95.html
- 1998: https://www.sfu.ca/~burton/ismm98.html
- 2000: http://www.cs.kent.ac.uk/events/conf/2000/ismm2000/
- 2002: http://www.hpl.hp.com/personal/Hans_Boehm/ismm/
- 2004: http://www.research.ibm.com/ismm04/
- 2006: https://www.cs.technion.ac.il/~erez/ismm06/
- 2007: http://www.eecs.harvard.edu/~greg/ismm07/
- 2008: http://www.cs.kent.ac.uk/~rej/ismm2008
- 2009: http://sysrun.haifa.il.ibm.com/hrl/ISMM2009/
- 2010: https://web.archive.org/web/20110215103030/http://www.cs.purdue.edu/ISMM10/

International Symposium on Memory Management II

- 2011: http://www.hpl.hp.com/personal/Hans_Boehm/ismm11/
- 2012: http://ismm12.cs.purdue.edu/
- 2013: https://www.cs.technion.ac.il/~erez/ismm13/
- 2014: http://ismm2014.cs.tufts.edu/
- 2015: http://conf.researchr.org/home/ismm-2015
- 2016: http://conf.researchr.org/home/ismm-2016
- 2017: https://conf.researchr.org/home/ismm-2017
- 2018: https://conf.researchr.org/home/ismm-2018
- 2019: https://conf.researchr.org/home/ismm-2019
- 2020: https://conf.researchr.org/home/ismm-2020

International Symposium on Memory Management 2020

Название докладов:

- Garbage Collection Using a Finite Liveness Domain
- Prefetching in Functional Languages
- Improving Phase Change Memory Performance with Data Content Aware Access
- ThinGC: Complete Isolation With Marginal Overhead
- Verified Sequential Malloc/Free
- Alligator Collector: A Latency-Optimized Garbage Collector for Functional Programming Languages
- Understanding and Optimizing Persistent Memory Allocation
- Exploiting Inter- and Intra-Memory Asymmetries for Data Mapping in Hybrid Tiered-Memories

ОО реализация

- Предлагаемый мною подход принципиально отличается от вышеизложенных и основан на реализации ООП в **C++**.
- Ограничить использование указателей на внутреннею структуру класса **C++** объявив их в закрытой области видимости. При правильной технологии ООП, а это свойство называется **инкапсуляцией**, это и так надо делать, пользователь не должен видеть как реализован класс, он должен использовать только его интерфейс.
- Во-вторых, в **C++** имеется выделенный **конструктор копирования**, который позволяет построить копию объекта. Осталось для выбранного класса определить функцию **swap**, для подмены объекта на его копию, и можно организовывать динамическое перераспределения памяти на **C++**.
- В отличии от небольших реализаций в 11 000 строк кода, реализация ОО подхода требует около 300 строк, включая вспомогательные шаблоны **C++**.

Пример использования

```cpp
#include "allocator.h"

class List {
  struct Node {
    int      mData;
    Node*    mNext;

    Node(int data, Node* next=NULL):
        mData(data),
        mNext(next) {
    }
    ~Node() {}
  };

  Allocator*  mAllocator;
  Node*       mHead;
```

Основное отличие от обычного кода строчка **15**, где содержится объявление переменной указателя на тип **Allocator**.

```
18 public:
19   List(Allocator *allocator):
20   …
21   List(const List& a, Allocator *allocator):
22   …
23       *mLink = new(mAllocator) Node(i, *mLink);
24   …
25       Node* tmp=*mLink;
26       *mLink = tmp->mNext;
27       mAllocator->destroy(tmp);
28   …
29   void swap(List &a) {
30     Allocator *tmp1=mAllocator;
31     mAllocator = a.mAllocator;
32     a.mAllocator = tmp1;
33
34     Node* tmp2 = mHead;
35     mHead = a.mHead;
36     a.mHead = tmp2;
37   }
```

В строках **23** и **27** показано его использование для выделения и освобождения памяти. Освобождение использует шаблонную функцию **destroy**, поскольку перегрузку стандартого **delete** дополнительными аргументами позволяют не все компиляторы **C++**.

```
18 public:
19   List(Allocator *allocator):
20   …
21   List(const List& a, Allocator *allocator):
22   …
23       *mLink = new(mAllocator) Node(i, *mLink);
24   …
25       Node* tmp=*mLink;
26       *mLink = tmp->mNext;
27       mAllocator->destroy(tmp);
28   …
29   void swap(List &a) {
30     Allocator *tmp1=mAllocator;
31     mAllocator = a.mAllocator;
32     a.mAllocator = tmp1;
33
34     Node* tmp2 = mHead;
35     mHead = a.mHead;
36     a.mHead = tmp2;
37   }
```

Для базовых типов и классов, для которых в принципе не нужен вызов деструктора, определена шаблонная функция **dealloc**. **destroy** и **dealloc** могут также дополнительно вызываться с дополнительным целочисленным аргументом для удаления массивов.

```
18 public:
19   List(Allocator *allocator):
20   …
21   List(const List& a, Allocator *allocator):
22   …
23       *mLink = new(mAllocator) Node(i, *mLink);
24   …
25       Node* tmp=*mLink;
26       *mLink = tmp->mNext;
27       mAllocator->destroy(tmp);
28   …
29   void swap(List &a) {
30     Allocator *tmp1=mAllocator;
31     mAllocator = a.mAllocator;
32     a.mAllocator = tmp1;
33 
34     Node* tmp2 = mHead;
35     mHead = a.mHead;
36     a.mHead = tmp2;
37   }
```

В строчках **21** и **29** содержатся заголовки **конструктора-копирования** и функции **swap**, которые необходимы для замены текущего объекта на его копию. Если Вы хотите для своего класса использовать оператор $=$, то его тоже необходимо реализовать. Он, как известно, не наследуется, но поддержка в классе шаблона **GC** имеется.

```
41 typedef GC<List> GCList;
42 …
43 GCList a;
44 …
45 a.reallocate();
46 …
```

На строке **41** объявляем новый тип **GCList**. Затем вызываем на строке **43 конструктор по умолчанию**. Предположим на строке **44** производим массовые операции по выделению/освобождению памяти. Память выделяется блоками, кратными размеру **страницы памяти** (Виртуальная память), затем раздается внутри конкретного объекта класса **List**. Если процент не используемой памяти высок, то вызов **reallocate** на строке **45** приведет к построению копии объекта, с его последующей **подменой** на первоначальную переменную.

```
41 typedef GC<List> GCList;
42 …
43 GCList a;
44 …
45 a.reallocate();
46 …
```

Естественно, вызов **reallocate** может происходит неоднократно, согласуясь с логикой программы. Сам вызов **reallocate** ничего не стоит, если не строится копия, т.е. его можно использовать практически при любом изменении объекта.

Также, попутно выполняются учет **работы времени GC**, **текущая память**, **максимальная память**, **прерывание программы при достижении заданного размера использованной памяти** и в **debug** версии программы **проверки на утечки памяти**. В отличии от malloc/free, проверка утечек памяти проводится для конкретного **Allocator**, что позволяет локализовать ошибку и быстро ее обнаружить.

Замена **alloca**

```
{
Allocator a[1];
…
List lst(a);
…
}
```

Если определять **Allocator** вначале блока и затем его использовать для работы с автоматическими переменными, использующими **Allocator**. После закрытия блока вся занятая ими память будет возвращена в систему.

Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz, cache size: 8192 KB

Cyclic7 (algorithm TQ)

	time	GC	GC/time %
Allocator	33.49 sec	0.11 sec	0.33%
Allocator (except for Integer)	33.88 sec	0.12 sec	0.35%
malloc/free	57.57 sec	—	—

Задача Попова 21 (algorithm TQ)

	time	GC	GC/time %
Allocator	545.78 sec	4.16 sec	0.76%
Allocator (except for Integer)	482.49 sec	1.17 sec	0.24%
malloc/free	498.68 sec	—	—

eco11 (algorithm TQ)

	time	GC	GC/time %
Allocator	200.85 sec	0.44 sec	0.22%
Allocator (except for Integer)	268.78 sec	0.07 sec	0.03%
malloc/free	341.10 sec	—	—