Лекция 7. Garbage Collectors

ИУ8

October 13, 2016

Содержание

На текущей лекции рассмотрим:

- Механизм garbage collection

Ремарка

Сборка мусора была впервые применена Джоном Маккарти в 1959г в языке программирования Lisp. В данный момент многие АЯП обладают механизмом сборки мусора. Поэтому на текущей лекции будут рассмотрены основные принципы систем сборки мусора

Garbage Collection

Сборка мусора одна из форм автоматического управления памятью.

Сборка мусора — технология, позволяющая, с одной стороны, упростить программирование, избавив программиста от необходимости вручную удалять объекты, созданные в динамической памяти, с другой — устранить ошибки, вызванные неправильным управлением памятью вручную.

Автоматическая сборка мусора передает ответственность за процесс освобождения памяти от объектов, которые больше не используются в ПО, от разработчика к среде испольнения программы

За и против ручного управления памятью

+ первое и главное "за" - эффективность механизма
+ контроль над выделением и освобождением памяти
- большая вероятность ошибки (утечка памяти, двойное освобождение памяти, обращение к освобожденной памяти)

За и против автоматической сборки мусора

+ низкая вероятность ошибки (утечка памяти, двойное освобождение памяти, обращение к освобожденной памяти)
- снижение производительности
- потеря контроля над выделением и освобождением памяти
- недетерминированность программы

Принцип работы GC

- Программист создает переменные, выделяет память и т.д.
- В произвольный момент времени (зависит от GC) система сборки мусора обнаруживает неиспользуемые объекты и неиспользуемую память
- Система сборки мусора освобождает всю неиспользуемую память и доступные объекты

Существует несколько алгоритмов, алгоритмов обнаружения GC неиспользуемых объектов

Алгоритмы сбора мусора

- Подсчет ссылок
- Маркировка и чистка
- Копирование

Посчет ссылкок

Старый добрый shared_ptr<T>

Копирование

Свободная память делится на две области: активную и неактивную.

В процессе сбора мусора объекты из активной области копируются в неактивную

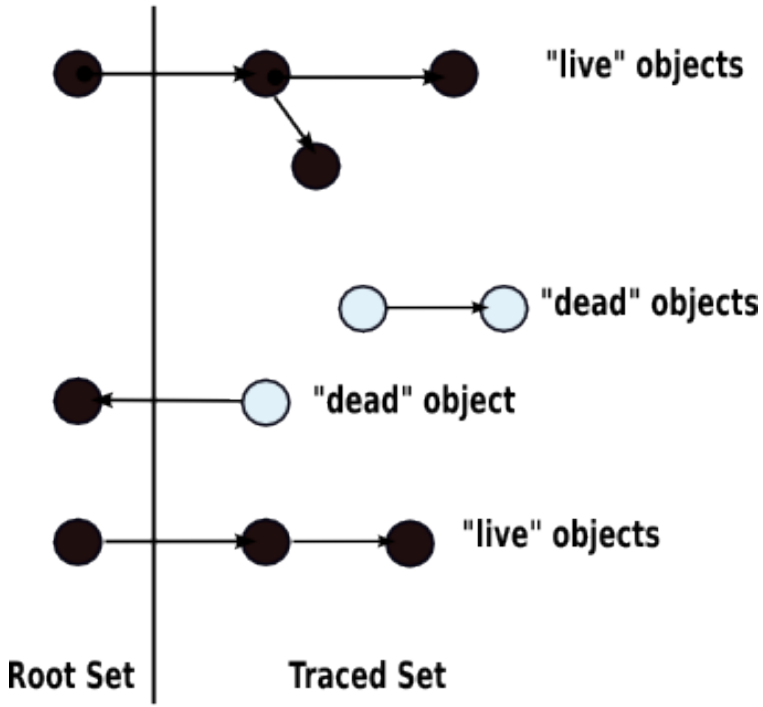
Неактивная область становится активной, и наоборот

Маркировка и очистка

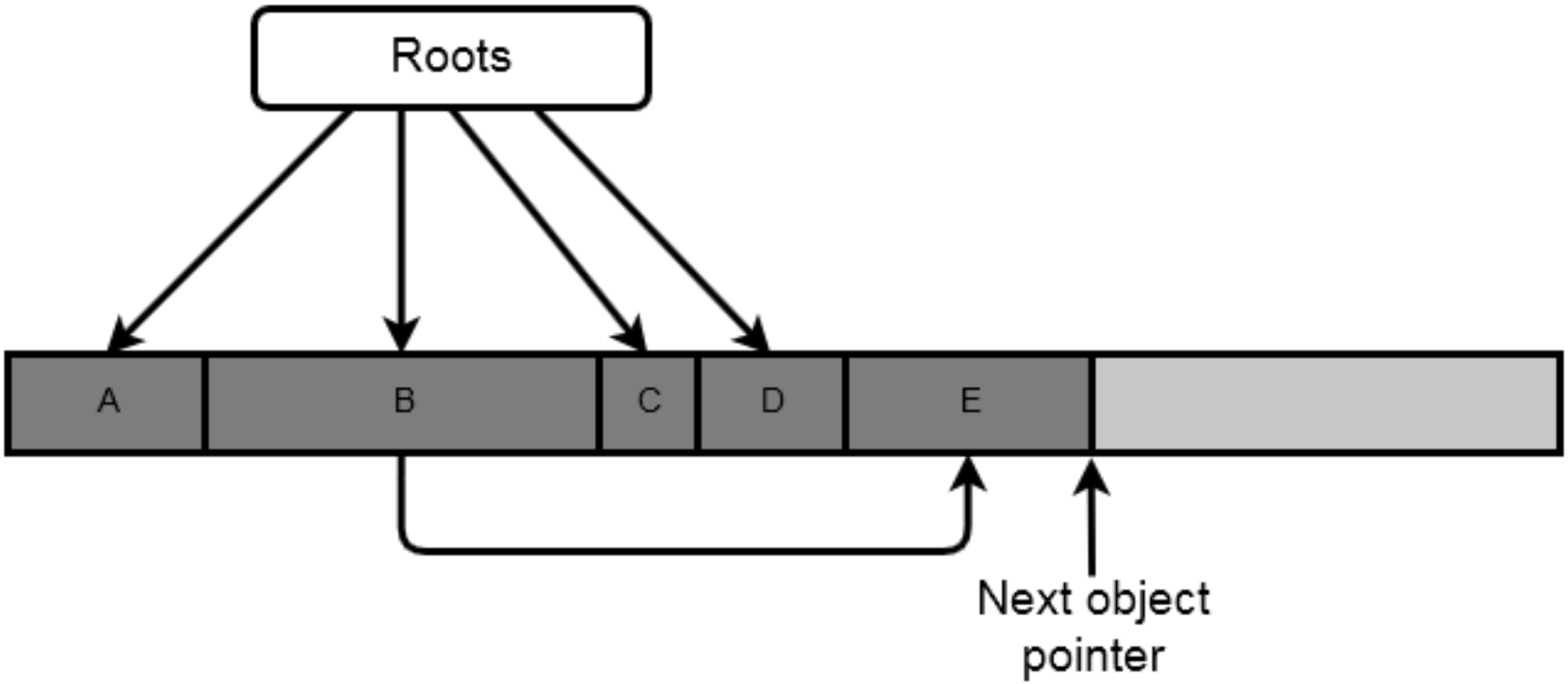
Алгоритм выполняется в два этапа **mark** и **sweep**

Во время маркировки каждый достижимый объект помечается битом достижимости

Во время очистки все объекты, не помеченные битом достижимости, удаляются

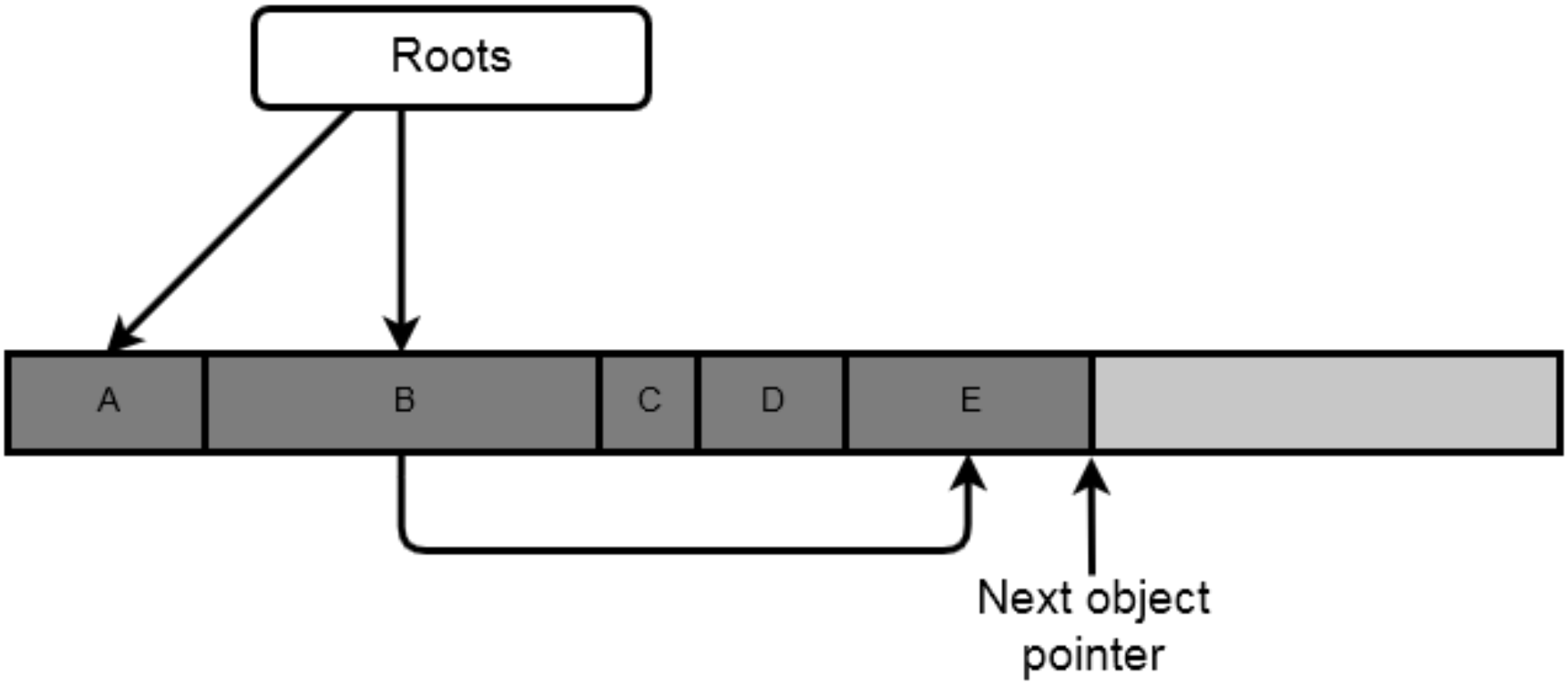


Рассмотрим работу GC с алгоритмом маркировки и очистки

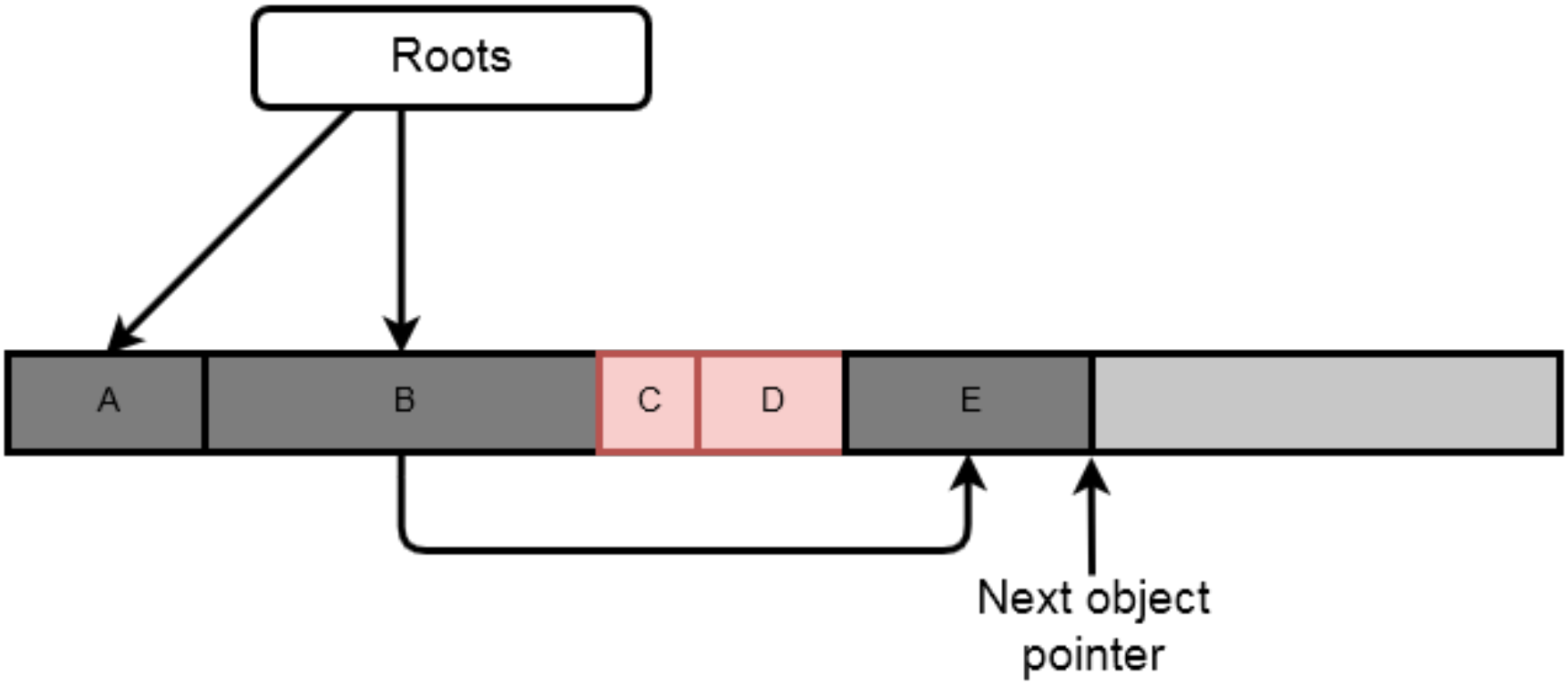


Все объекты достижимы.

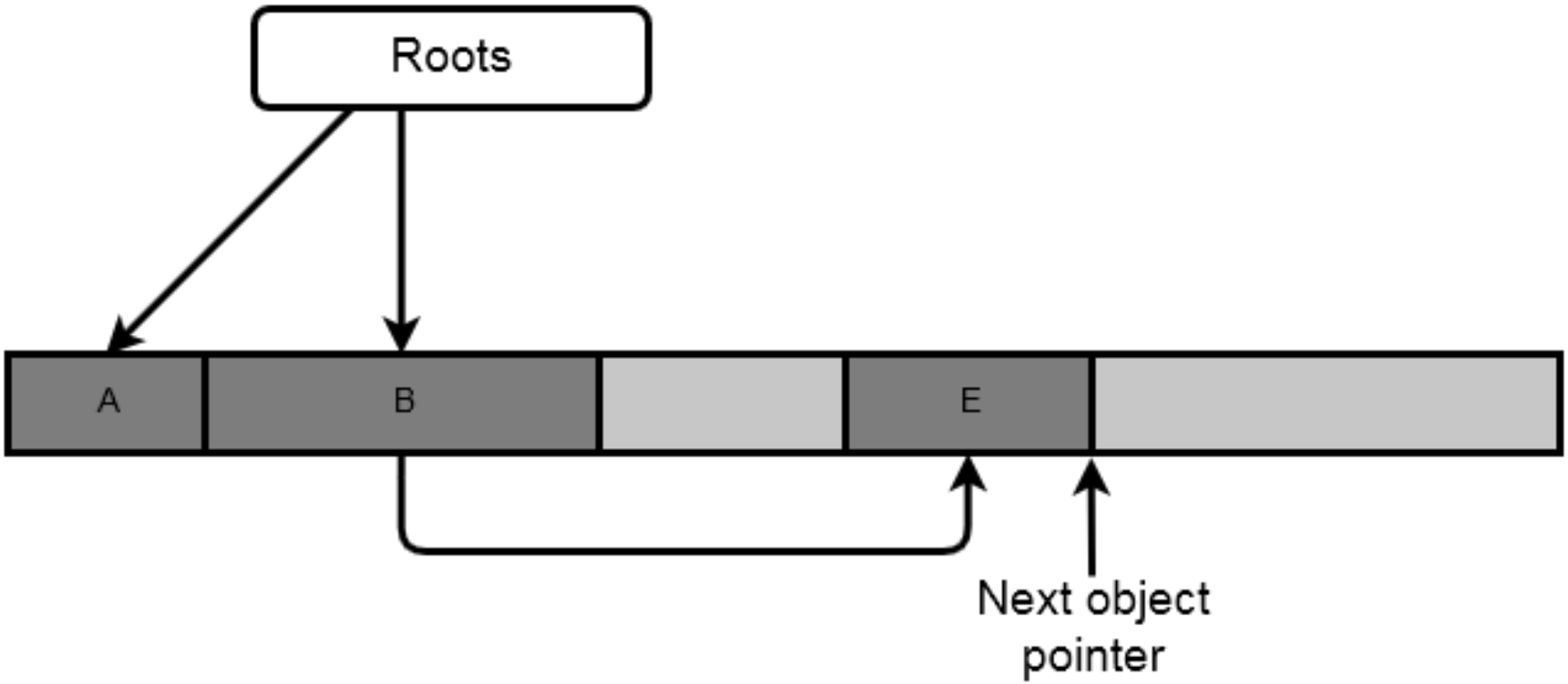
По той или иной причине, объекты C и D перестали быть достижимыми.



GC начинает процесс сборки мусора. Он проходит по всем объектам и определяет, какие из них достижимые. Объектам A, B, E выставлены биты достижимости. У объектов C и D биты достижимости не установлены.

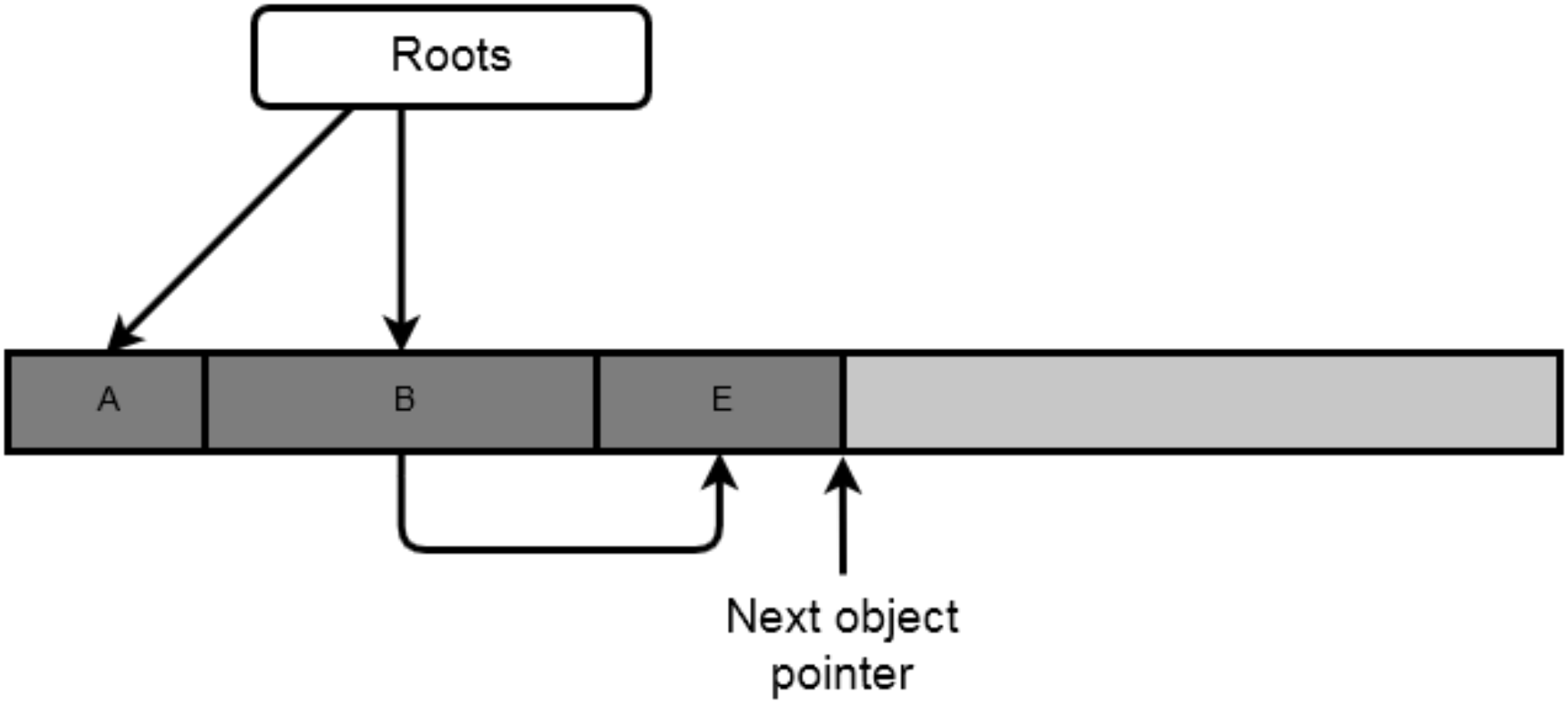


Запускается второй этап сборки мусора - очистка. Все недостижимые объекты удаляются.



На этом этап сборки мусора можно завершать.

Однако, для ускорения выделения памяти для нового объекта, память может подвергаться дефрагментации



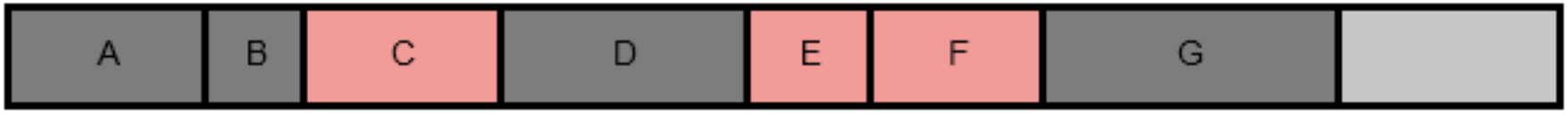
Поколения в системах сборки мусора

В ходе многолетних разработок систем сборки мусора стало очевидно, что недавно созданные объекты становятся недостижимыми чаще, чем объекты, существующие длительное время.

Это послужило поводом к созданию сборщиков мусора с поддержкой поколений объектов

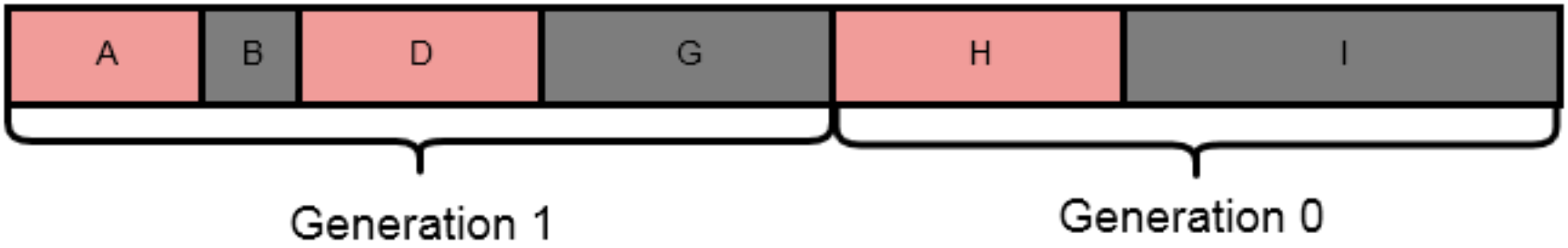
Скорость сборки мусора в таких системах выше

В начальный момент все элементы принадлежат нулевому поколению

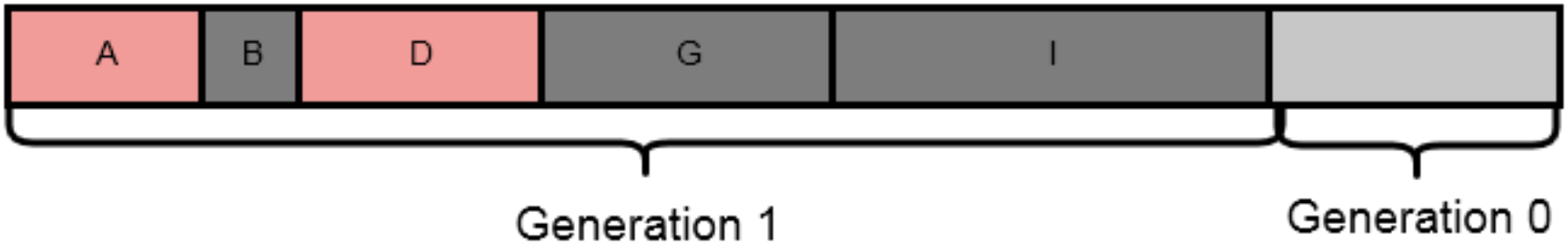


Все элементы, пережившие сборку мусора, переходят в следующее поколение

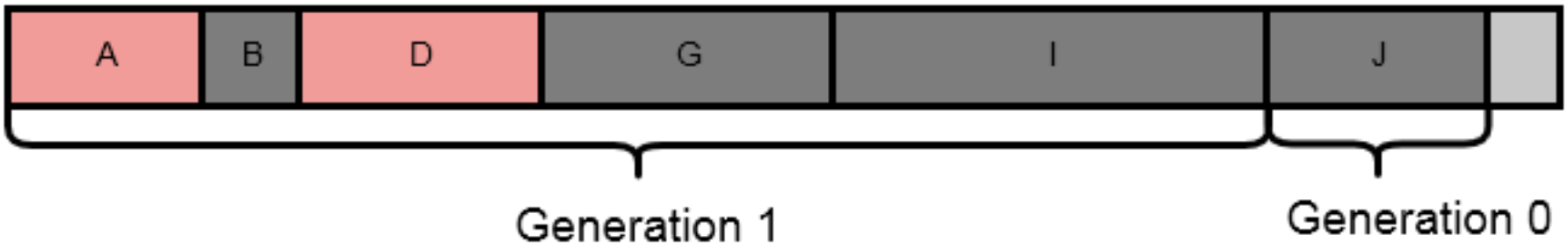
Новые элементы добавляются в нулевое поколение



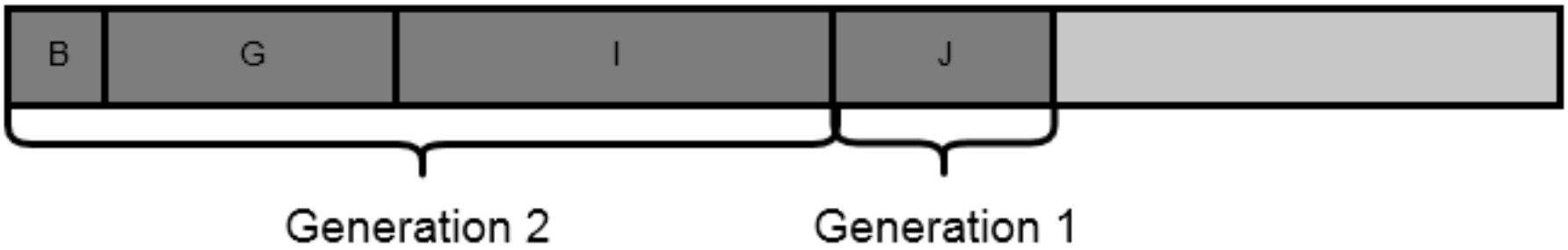
Если условия сборки мусора не выполняются для первого поколения, то сборка выполняется только в нулевом поколении



Новые элементы добавляются в нулевое поколение



В какой-то момент времени происходит очистка и первого и нулевого поколения. Объекты пережившие сборку переходят в следующие поколения



What about C++ Standart

Сборка мусора является опциональна в C++, т.е. отдается на усмотрение разработчиков компиляторов реализовывать GC или нет.

Именно по этой причине, стандарт C++11 обеспечивает некий контроль над действиями GC.

Problem

```
1 int* p = new int;
2 p+=10;
3   // ... collector may run here ...
4 p-=10;
5 *p = 10; // can we be sure that the int is still there?
```

Solution

Память по адресу **p** не должна освобождаться GC

```
1 void declare_reachable(void* p);
```

Аннулирует действие declare_reachable

```
1 template<class T>
2 T* undeclare_reachable(T* p);
```

Чтобы память по адресу **p** стала вновь "недостижимой", функция undeclare_reachable должна быть вызвана столько раз, сколько была вызвана функция declare_reachable

Performance

```
1 void declare_no_pointers(char* p, size_t n); // p[0..n] holds no pointers
2 void undeclare_no_pointers(char* p, size_t n);
```

Функция declare_no_pointers предупреждает GC, что память p[0..n] не содержит отслеживаемых указателей. Таким образом, GC перестает рассматривать память p как нечто, что может содержать указатели на отслеживаемые объекты.

Функция undeclare_no_pointers аннулирует действие declare_no_pointers.

A garbage collector for C and C++

В качетсве примера GC, который работает с кодом, написанным на C и C++, можно привести Boehm collector

Boehm-Demers-Weiser garbage collector можно использовать как систему сборки мусора. Механизм заменяет **malloc** и **new**. Boehm collector позволяет выделять память и не заботиться об освобождении этой памяти. Boehm collector автоматически освободит память, когда определит, что она больше не доступна из программы.

Сборщик мусора работает по алгоритму "маркировка и очистка". Также он позволяет выполнять финализаторы (некоторые функции, которые выполняются перед удалением неиспользуемых объектов).

У Boehm GC есть режим работы, в котором он выполняет функцию системы обнаружения утечек памяти (leak detector).

Using Boehm GC

```
1  #include "gc.h"
2  #include <assert.h>
3  #include <stdio.h>
4
5  int main() {
6    GC_INIT();
7    for (int i = 0; i < 10000000; ++i)
8    {
9       int **p = (int **) GC_MALLOC(sizeof(int *));
10      int *q = (int *) GC_MALLOC_ATOMIC(sizeof(int));
11      assert(*p == 0);
12      *p = (int *) GC_REALLOC(q, 2 * sizeof(int));
13      if (i % 100000 == 0)
14       printf("Heap size = %d\n", GC_get_heap_size());
15   }
16   return 0;
17 }
```

gcpp: Deferred and unordered destruction

gcpp не является полноценной системой сборки мусора в традиционном пронимании этого термина. gcpp скорее можно описать как механизм управления объектами, нацеленный на отложенное удаление объектов.

Несмотря на это, gcpp использует принципы и механизмы, присущие классическим сборщикам мусора

This is a demo of a potential additional fallback option for the rare cases where unique_ptr and shared_ptr aren't quite enough, notably when you have objects that refer to each other in local owning cycles, or when you need to defer destructor execution to meet real-time deadlines or to bound destructor stack cost. The goal is to illustrate ideas that others can draw from, that you may find useful even if you never use types like the ones below but just continue to use existing smart pointers and write your destructor-deferral and tracing code by hand.

deferred_heap

Класс deferred_heap владеет памятью, содержащей объекты, к которым можно обращаться с использованием deferred_ptr<T>

Чтобы создать новый объект, необходимо использовать метод make<T>(). Если тип T имеет деструктор, он будет вызван при сборке недостижимых объектов.

Класс deferred_heap предоставляет метод collect(), вызов этого метода запускает освобождение памяти от недостижимых объектов и запуск их деструкторов.

deferred_ptr

Доступ к объектам, расположенным в deferred_heap, осуществляется через класс deferred_ptr<T>

deferred_ptr предоставляет такой же функционал, как и любой другой "умный" указатель в C++.

Каждый объект класса deferred_ptr связан с конкретным объектом deferred_heap, поэтому присвоение разных deferred_ptr, относящихся к разным deferred_heap, запрещено

Когда deferred_ptr полезен

Использование gcpp оправдано, когда требуется управление памятью в структурах со сложными циклическими связями.

Также gcpp-подобные системы имеет смысл использовать в системах реального времени.

Использование shared_ptr и unique_ptr в системах реального времени может быть ограничено по причине "несвоевременных" вызовов деструкторов. Проблема заключается в том, что при использовании shared_ptr деструктор объекта может выполняться во время работы участка кода, требующего максимальную производительность.

Функционал отложенных вызовов деструкторов, предоставляемый deferred_heap, будет полезным при решении описанной проблемы.

В качестве заключения

Память приложения является одним из главных ресурсов. Грамотная работа с ней является одной из основных метрик качества ПО. Неумелое использование памяти приводит к ошибкам и неоправданным затратам ресурсов.

Трудно найти другой язык программирования, обладающий такой же гибкостью в области управления памятью, как C++. C++ предоставляет такие средства управления памятью, как аллокаторы и интелектуальные указатели. Все эти средства могут быть расширены под любые требования. Кроме того, используя C++, можно реализовать другие системы и механизмы управления памятью, выходящие за рамки стандартных средств.

The end

Самостоятельное изучение

- gccp repo
- GC Algorithmic Overview
- Garbage collection ABI

Список литературы

- gccp repo
- Boehm-Demers-Weiser Garbage Collector
- Boehm Collector site
- Garbage Collection in the C++ Standard
- Bjarne Stroustrup - The C++ Programming Language