МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

Кафедра дискретной математики и алгоритмики

ХОДОР
Иван Андреевич

РАЗРАБОТКА И РЕАЛИЗАЦИЯ МЕТОДОВ АВТОМАТИЧЕСКОГО УПРАВЛЕНИЯ ПАМЯТЬЮ

Дипломная работа

Научный руководитель:
старший преподаватель
И. Д. Лукьянов

Допущена к защите
«____»_____ 2022 г.
Зав. кафедрой дискретной математики
и алгоритмики, кандидат физ.-мат. наук,
профессор В. М. Котов

Минск, 2022

СОДЕРЖАНИЕ

РЕФЕРАТ

Дипломная работа, 66 стр., 3 иллюстр., 46 источников.

Ключевые слова: C++, АЛЛОКАТОР, ВЫДЕЛЕНИЕ ПАМЯТИ, УДАЛЕНИЕ ПАМЯТИ, КЛАСС, ОБЪЕКТ, СБОРКА МУСОРА.

Объект исследования — методы работы с памятью в C++, методы сборки мусора в различных языках программирования.

Цель работы — реализация сборщика мусора для языка программирования C++.

Работа посвящена разработке и реализации сборщика мусора для языка программирования C++ и анализ полученного решения.

РЭФЕРАТ

Дыпломная праца, 66 с., 3 рыс., 46 крыніц.

Ключавыя словы: С++, АЛЛАКАТАР, ВЫЛУЧЭННЕ ПАМЯЦІ, ВЫДАЛЕННЕ ПАМЯЦІ, КЛАСС, АБ'ЕКТ, ЗБОРКА СМЕЦЦЯ.

Аб'ект даследавання — метады працы з памяццю ў С++, метады зборкі смецця ў розных мовах праграмавання.

Мэта працы — рэалізацыя зборшчыка смецця для мовы праграмавання С++.

Праца прысвечана распрацоўцы і рэалізацыі зборшчыка смецця для мовы праграмавання С++ і аналіз атрыманага рашэння.

ABSTRACT

Diploma thesis, 66 p., 3 fig., 46 sources.

Keywords: C++, ALLOCATOR, MEMORY ALLOCATION, MEMORY DEALLOCATION, CLASS, OBJECT, GARBAGE COLLECTION.

The object of research is a methods of working with memory in C++, garbage collection methods in various programming languages.

Objective: implement a garbage collector for the C++ programming language.

The work is devoted to the development and implementation of a garbage collector for the C++ programming language and the analysis of the resulting solution.

ВВЕДЕНИЕ

В современном мире инженеры-программисты редко занимаются непосредственным контролем за логикой управления памятью, т.к. чаще всего стоит сосредоточиться на более важных вещах вроде бизнес-логики приложений. Обычно существуют готовые инструменты, которые занимаются эффективным распределением памяти исходя из нужд разработчиков. Самым популярным методом автоматического управления памятью являются сборщики мусора. Для качественной разработки сборщика мусора на языке программирования C++ (что является целью данной работы) стоит изучить методы управления памятью, которые используются в этом языке на текущий момент. Одной из наиболее важных концепций в этом направлении являются аллокаторы, которые являются прослойкой между разработчиком, использущим различные инструменты (например структуры данных), и низкоуровневыми функциями выделения/освобождения памяти. Аллокаторы позволяют инкапсулировать всю логику организации памяти для достижения необходимых нужд.

После понимания структуры такого понятия как аллокатор можно приступать к изучению более сложных методов управления памятью. Сборка мусора – популярный во многих языках программирования приём, который, тем не менее, является довольно нетривиальным. Написание своего сборщика мусора довольно интересная техническая задача на любом языке программирования, однако тот факт, что цель работы сделать это на C++, вносит ещё больше сложностей технического характера в силу специфики этого языка.

ГЛАВА 1

КОНЦЕПЦИЯ АЛЛОКАТОРОВ

1.1 Развитие стандартного аллокатора и его недостатки

Аллокатор (англ. allocator) или распределитель памяти — специализированный класс или набор методов, реализующий и инкапсулирующий малозначимые с прикладной точки зрения детали распределения и освобождения ресурсов компьютерной памяти. Изначально аллокаторы являются концепцией языка программирования C++, потому все примеры будут рассматриваться именно на этом языке.

Стоит отметить, что под аллокатором можно понимать как все способы управления динамической памятью, так и конкретно `std::allocator`. В данной работе используется первый более общий вариант.

Для понимания текущего устройства аллокаторов и проблем в их архитектуре необходим проследить историю их развития.

1.1.1 `malloc/free`

Первым стандартным способом управления динамической памятью являлась пара методов `malloc/free`:

```
void* malloc(size_t size);
void free(void* p);
```

`malloc` выделяет указанное количество байт и возвращает `void*`, что приводит к проблемам с приведением типов и возможным неопределённым поведением.

В этих методах наблюдается некоторая асимметрия: `malloc` знает размер выделяемого блока памяти, в то время как `free` нет. Из-за такого ABI – бинарного интерфейса приложения, англ. application binary interface – компиляторы обязаны сохранять размер блока памяти, выделенного по указателю, что влияет на эффективность разрабатываемых приложений. Однако пользователь всегда знает размер выделяемой и удаляемой памяти, значит может передавать этот размер в указанные методы, что могло бы повысить эффективность программы. Тем не менее, в C++ ABI является устоявшимся, потому в данный момент исправления в этом направлении не предусматриваются. Можно сделать вывод, что аллокатор должен следить за размерами всех блоков памяти, которыми управляет. Однако менеджмент размеров блоков памяти вносит неуместные сложности и проблемы с производительностью в архитектуру любого аллокатора, т.к. это довольно сложная задача, при решении которой легко упустить множество важных мелочей. В подтверждение приведём перевод цитаты из предложения – более принятым является английский термин proposal – в стандарт C++ под номером N3536 [20]:

> *... упущение [информации о размере блока памяти в глобальном `operator delete`(он же `free`)] имеет, к сожалению, неприятные для эффективности последствия. Современные аллокаторы обычно работают в категориях размеров блоков памяти, и, для сохранения эффективности работы с данными, не могут хранить информацию о размерах рядом с аллоцируемыми объектами, из-за чего деаллокация блока требует поиска размера этого самого блока. Этот поиск может быть неэффективен, частично из-за того, что поисковые структуры данных в большинстве своём не являются кеш-дружественными...*

Одним из предлагаемых решений является следующее:

```
struct Blk {void* ptr; size_t length;};
Blk malloc(size_t size);
void free(Blk block);
```

1.1.2 operator new/operator delete

Следующим появился `operator new`:

- Работает с `malloc` (ради обратной совместимости);
- Использует информацию о конструируемом типе. Теперь можно передавать не количество байт, а количество объектов, что является наиболее значимым нововведением. Более того, результатом вызова метода является не `void*`, как у `malloc`, а `T*`, что позволяет избежать некоторых проблем с неопределённым поведением при приведении типов;
- Добавлено некоторое количество синтаксических улучшений для удобства его использования. Например, результатом `new int` будет являться `int*`, но результатом `new int[100]` также `int*`, несмотря на то, что `int` и `int[100]` – разные типы;
- "The Great Array Distraction". Это правило гласит, что в случае, когда вы вызываете `operator new` для удаления, вы обязаны использовать `operator delete`, но когда вызываете `operator new[]`, вы должны использовать `operator delete[]`, иначе возможны проблемы с утечками памяти и неопределённым поведением. Проблема в том, что для этого правила нет никаких причин: это просто некоторая договорённость, которой решили следовать.

Для удобства существует множество различных версий этих методов: `new`, конструирующий объект из переданных аргументов; `placement new`, позволяющий конструировать объект по переданному указателю; `nothrow new`, который, в отличие от других версий, в случае проблем не будет бросать исключения, а вернёт `nullptr`. Также при необходимости можно перегружать свои версии. Главное помнить, что делать это всегда нужно парами.

Однако, несмотря на всё разнообразие возможностей, предоставляемых `new`/`delete`, при таком подходе существуют очевидные проблемы. Рассмотрим следующий код:

```
int* ptr = new int;
// some code here
delete ptr;
```

Несмотря на то, что случай тривиален, ошибиться довольно просто в случае, когда между выделением и удалением памяти расположены сотни или тысячи строк: разработчику очень легко забыть удалить выделенную память, что приводит к утечкам памяти. Это небезопасно и неэффективно.

Также существуют гораздо более сложные сценарии:

```
int* get_ptr() {
  return new int;
}
...
int* ptr = get_ptr();
```

Разработчику сложно понять, кто несёт ответственность за удаление нового объекта. Для этого придётся среди всего множества кода искать вызываемую функцию, разбираться в её поведении, просматривать связаный код на случай, если удаление производится в другом месте. Подобные действия значительно замедляют процесс написания новой логики. И даже если разработчик ошибётся, считая, что он может удалить объект, повторное удаление приведёт к неопределённому поведению.

Использование `operator new`/`operator delete` в современном C++ не рекомендуется к использованию при отсутствии серьёзных на то причин.

1.1.3 `std::allocator`

Современная версия аллокатора в C++ – `std::allocator`. Он берёт на себя несколько задач: выделение/удаление памяти под объект, вызов конструктора/деструктора объекта.

Среди известных участников комитета по развитию C++ бытует мнение, что с архитектурой `std::allocator` есть некоторые проблемы:

> *`std::allocator` не предназначен для аллоцирования памяти. Это как квадратное колесо у автомобиля с большим количеством*

колёс. Большинство разработчиков говорят, что это не является проблемой, и, возможно, это правда, ведь он всё ещё существует в текущей форме и не был удалён/изменён в последних стандартах. Андрей Александреску.

Настоящая причина возникновения `std::allocator`(что произошло в 1994 году [1]) – это необходимость некоторого механизма, который позволил бы избавить разработчика от взаимодействия с `near/far` указателями. Позже комитет убрал из языка всё с ними связанное, концептуально ничего не меняя в `std::allocator`. Это является огромной дырой в архитектуре современных аллокаторов, **потому что они не были спроектированы для аллоцирования изначально**.

Вот некоторые проблемы в архитектуре аллокаторов:

1. Тип как аргумент аллокатора.

   Аллокатор не должен совершать много действий с типом. По факту для осуществления эффективных операций он должен использовать лишь два значения: размер требуемого блока памяти и выравнивание. Например, разработчик как пользователь скорее всего не хочет различных способов аллоцирования памяти для `int` и `unsigned int`, как и не захочет различных способов аллоцирования памяти для типа `T1` размером 128 байт и типа `T2` размером также 128 байт. Аллокатор, который будет хорошим для `T1`, скорее всего подойдёт и для `T2`. Он будет хорошим для этого конкретного размера и выравнивания, но не для конкретного типа.

2. Аллокатор должен **аллоцировать** память, но он также выступает в роли паттерна фабрика: конструирует объекты (с C++20 это исправлено и методы `construct/destroy` вынесены из `std::allocator` как `std::construct_at/std::destroy_at` [26][27], однако 20й стандарт ещё не широко распространён, потому пока будем считать что проблема есть).

   С другой стороны это можно воспринимать как плюс, ведь стандартный аллокатор разделяет операции выделения памяти и конструирования объекта(в отличие от `new`), и теперь разработчик может сам выбирать, какое поведение ему стоит переопределить, а какое нет.

3. Несмотря на то, что `std::allocator` имеет некоторые возможности фабрики и знание о типе, он всё ещё использует `void*`.

4. Опять же, проблема с оперированием размерами блоками памяти. Лучше было бы, если бы они работали с `Blk`.

5. Наличие в `std::allocator` `rebind<U>::other` для получения такого же аллокатора, но для другого типа. Стоит отметить, что с C++17 эта часть стандартной библиотеки помечена устаревшей и будет удалена в C++20 [2]. Такое решение было принято в силу того, что `std::allocator_traits` стал брать на себя больше обязанностей, из-за чего `rebind` стал избыточным в `std::allocator`.

Однако, несмотря на все недостатки, `std::allocator` неплохо справляется со своими обязанностями, пусть и с некоторыми неудобствами. Он помогает абстрагироваться от модели адресации памяти, позволяя разработчику сконцентрироваться на более важных вещах.

Существует разделение аллокаторов на statefull (имеющих некоторое состояние) и stateless (не имеющих состояния). Хранение некоторого состояния может быть полезно при сложной логике работы с памятью: например, хранение некоторой метаинформации или использование общего буфера. Statefull аллокаторы были запрещены до C++11. К счастью, это было исправлено, что позволило реализовать широкое множество эффективных стратегий.

1.2 Общие принципы

Как уже упоминалось, `std::allocator` занимается выделением/удалением памяти и вызовом конструктора/деструктора объекта.

Для этого существуют методы `allocate/deallocate` и `construct/destroy` соответственно [2]. В простейшем варианте стандартный аллокатор, выделяющий память для объектов типа `T`, устроен так:

```
namespace std {
template <class T>
```

```
struct allocator {
  void deallocate(T* p, size_t) { ::operator delete(p); }

  T* allocate(size_t n) {
    return reinterpret_cast<T*>(::operator new(n * sizeof(T)));
  }
}
} // namespace std
```

В данном случае используются именно глобальные версии `new/delete`, т.к. требуется лишь выделить память, а вызов конструктора/деструктора возлагается на методы `construct/destroy`. Несмотря на то, что мы их не реализовали, специальный класс `std::allocator_traits<T>` проверит их наличие и реализует в случае их отсутствия [4]. Заметим, что в методе `deallocate` есть второй параметр типа `size_t`. Стандарт стал требовать передачу размера удаляемого блока памяти в аллокатор, однако это почти никак не помогает в силу того, что невозможно применить эту информацию методами `free/operator delete`. Однако её можно использовать для других целей.

Для понимания приведём реализации методов `construct/destroy`:

```
template <typename ...Args>
void construct(T* p, Args&& ...args) {
  ::new(static_cast<void*>(p)) T(std::forward<Args>(args)...);
}
void destroy(T* p) { p->~T(); }
```

1.2.1 std::allocator_traits

`std::allocator_traits` – вспомогательный класс, содержащийся в контейнерах стандартной библиотеки. Он выступает как прослойка между аллокатором и контейнером и предоставляет реализацию неообходимых частей аллокатора в случае их отсутствия. Например методы `construct/destroy`, алиасы для типов `allocator_type`, `value_type`, `pointer`, `size_type`, `rebind_alloc<T>`, `rebind_traits<T>` и множество других [3]. Ранее автор аллокатора должен был реализовывать это всё сам.

Для проверки наличия методов и алиасов для типов `std::allocator_traits` использует приём SFINAE(Substitution Failure Is Not An Error). Разъяснение этого способа находится за рамками данной работы.

1.2.2 Использование аллокаторов

Рассмотрим сигнатуру `std::vector` [5]:

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;
```

Как видим, вторым шаблонным параметром передаётся аллокатор, причём по умолчанию это `std::allocator`. Значит, для использования другого аллокатора нужно лишь передать его вторым аргументом в шаблон контейнера:

```
std::vector<int, my_allocator<int>> v;
```

Аналогично для других стандартных контейнеров.

ГЛАВА 2

ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ АЛЛОКАТОРОВ

2.1 Стандартные примеры нестандартных аллокаторов

Одной из проблем инструментов стандартной библиотеки является то, что они предназначены для общего использования. Они обязаны одинаково хорошо работать на размерах от 1 байта до 10Гб, в то время как, зная некоторые особенности своих нужд, разработчик может применять более оптимальные стратегии поведения. Потому существует множество нестандартных аллокаторов, применимых в различных ситуациях. Также частым приёмом при реализации нового аллокатора является применение композиции нескольких аллокаторов. Если взглянуть на документацию популярных аллокаторов [6][7], то можно заметить, что почти всегда они построены на применении нескольких абсолютно различных стратегий управления памятью в зависимости от некоторых условий. Далее рассматриваются самые популярные нестандартные аллокаторы и способы применения композиций.

2.1.1 Null-аллокатор

Самым простым примером нестандартного аллокатора является `null`-аллокатор [8]. На запрос выделить память он возвращает `nullptr`, а на запрос удаления памяти проверяет, является ли этот указатель `nullptr`, потому что аллокатор может принимать лишь то, что он возвращал.

`Null`-аллокатор часто используется в качестве терминальной стратегии. Представим, что разработчик спроектировал аллокатор, использующий 10 различных стратегий для различных размеров требуемой памяти. Но для очень

больших размеров он понятия не имеет, как лучше всего будет поступать. Использование `null`-аллокатора будет наиболее верным решением.

2.1.2 Pooled-аллокаторы

Полезной концепцией является `pooled`-аллокаторы: первоначально выделяется некоторый участок памяти, после чего все операции происходят только с ним.

Полезным может быть следующий приём: зная, что за всё время использования программы она суммарно не потребует более, например, 1Гб памяти, можно выделить блок такого размера в начале. Выделение будет происходить согласно некоторой стратегии, а удаление не будет происходить вообще[9], чтобы не тратить на него время. При достаточном количестве памяти и недостаточном количестве времени получится достичь большей эффективности. Аллокаторы с подобной стратегией называют monotonic-аллокаторами.

2.1.3 Fallback-аллокатор

Простейшей композицией является `fallback`-аллокатор [8]:

```
template <class Primary, class Fallback>
class FallbackAllocator : private Primary, private Fallback {
 public:
    Blk allocate(size_t);
    void deallocate(Blk);
};
```

По умолчанию такой аллокатор использует `primary`-стратегию, но в случае, если что-то пойдёт не так, `fallback`-стратегию.

Наследование особенно полезно в случае stateless аллокаторов, что позволяет компилятору задействовать `Empty Base Class Optimization` [10].

Метод `allocate`:

```
template <class P, class F>
Blk FallbackAllocator::allocate(size_t n) {
```

```
    Blk r = P::allocate(n);
    if (!r.ptr) r = F::allocate(n);
    return r;
}
```

В случае `deallocate` требуется решить проблему выбора аллокатора для удаления. Становится понятно, что интерфейса только с двумя методами выделения/удаления памяти недостаточно. Нужен ещё один: метод `owns`, который будет подсказывать, владеет ли аллокатор указателем. При таком дополнении реализовать `deallocate` становится просто:

```
template <class P, class F>
void FallbackAllocator::deallocate(Blk b) {
    if (P::owns(b)) P::deallocate(b);
    else F::deallocate(b);
}
```

Этот метод точно нужно реализовать `primary`-аллокатору. Интересно, что его необязательно реализовывать для `fallback`: в случае деаллокации он не требуется. Но правилом хорошего тона является реализовать метод `owns` и для всего `FallbackAllocator`:

```
template <class P, class F>
bool FallbackAllocator::owns(Blk b) {
    return P::owns(b) || F::owns(b);
}
```

Причём C++ имеет замечательную особенность: в случае, если `fallback`-аллокатор не реализует метод `owns` и `FallbackAllocator::owns` никогда не будет вызван, программа успешно скомпилируется и корректно отработает. Потому разработчик сам вправе решать, необходимо ли ему реализовывать требуемый метод в зависимости от своих потребностей.

Чаще всего в качестве `fallback`-аллокатора используется `malloc`.

2.1.4 Stack-аллокатор

Как известно, использование памяти на стеке гораздо быстрее выделения/удаления памяти в куче, потому при выполнении некоторых условий использование такого аллокатора может положительно сказаться на эффективности программы.

Рассмотрим `stack`-аллокатор [8]:

```
template <size_t s>
class StackAllocator {
    char d_[s];
    char* p_;
    StackAllocator() : p_(d_) {}
    ...
};
```

Реализация метода `owns` тривиальна:

```
bool owns(Blk b) {
    return d_ <= b.ptr && b.ptr < d_ + s;
}
```

Перед реализацией методов `allocate`/`deallocate` введём функцию `roundToAligned(n)`, которая округляет пришедший размер блока памяти к ближайшему выравненному значению.

Реализация метода `allocate`:

```
Blk allocate(size_t n) {
    auto n1 = roundToAligned(n);
    if (n1 > (d_ + s) - p_) {
        return {nullptr, 0}; // memory is not enough
    }
    Blk result = {p_, n};
    p_ += n1;
    return result;
}
```

Здесь осуществляется проверка, достаточно ли памяти с учётом выравнивания, чтобы выделить нужное количество байт, и в случае успеха возвращается новый блок.

Реализация метода `deallocate`:

```
void deallocate(Blk B) {
    if (b.ptr + roundToAligned(b.n) == p_) { p_ = b.ptr; }
}
```

В силу устройства аллокатора нет возможности удалять случайный блок памяти. Это возможно только в случае, если происходит удаление последнего блока: размер блока с выравниванием равен текущему указателю.

Удобной особенностью является возможность удалить всё с асимптотикой $O(1)$ по времени:

```
void deallocateAll() { p_ = d_; }
```

Однако использовать такое надо с осторожностью.

Учитывая, что размер памяти в нём ограничен, можно создать удобную композицию с использованием `fallback`-аллокатора:

```
template <size_t s>
class StackAllocator {...};
class Mallocator {...};
using MyAlloc = FallbackAllocator<
    StackAllocator<16384>,
    Mallocator
>;
```

2.1.5 Freelist

`Freelist` [8] всегда имеет основной аллокатор, который выделяет/удаляет память в большинстве случаев. Сам он вступает в дело, когда происходят деаллокации некоторого конкретного размера. В самом начале указатель на начало списка пуст, но в случае, когда удаляется подходящий блок, он просто добавляется в лист блоков. В следующий раз, когда требуется выделить блок какого-то

специфического размера и лист не пуст, берём блок оттуда. Это позволяет не тратить время на выделение памяти из системы.

Однако есть и недостатки: память, попавшая в лист свободных блоков, никогда не освободится. После выделения 10000 блоков по 64 байта получается ситуация, когда все они находятся в аллокаторе. Это приводит к высокой фрагментации памяти.

Также у `freelist` есть проблемы с выполнением во многопоточной среде: если блок пришёл в глобальный лист, никто не гарантирует, что через наносекунду его не заберёт другой поток. Для решения этой проблемы разработчики сначала реализуют `lock free list` как первый уровень защиты. Часто также каждому потоку создают отдельный `freelist`, а как запасной создают глобальный.

Также стоит быть осторожным с манипуляциями блоками памяти, т.к. в силу свойства интрузивности один блок может оказаться в нескольких листах. Это может привести к потере данных или двойному освобождению.

Но несмотря на все недостатки, `freelist` – довольно мощный паттерн, который очень хорош для объектов небольшого размера. Пусть он и очень сложный.

Стоить отметить один интересный момент в реализации `deallocate`:

```
template <class Alloc, size_t Size>
class Freelist {
    Alloc parent_;
    struct Node { Node* next; } root_;
    void deallocate(Blk b) {
        if (b.length != Size) return parent_.deallocate(b);
        auto p = (Node*)b.ptr;
        p.next = root_;
        root_ = p;
    }
};
```

Учитывая, что это свободная память, и данные в ней уже не нужны, разработчик может не создавать дополнительную память для хранения информации о

следующем блоке, ведь можно хранить её прямо в свободном блоке. Потому вся дополнительная память – это всего одно поле `root_`.

Некоторые улучшения такого аллокатора:

- Предоставление памяти в некотором диапазоне.

  Если используемый `freelist` хорош в управлении блоками размером 1Кб, то имеет смысл отдавать блок при запросе меньше 1Кб. Возможно, получится добиться некоторого прироста эффективности. Потому иногда устанавливаются некоторые границы (в указанном примере можно отдавать блок при запросе в диапазоне от 513 байт до 1Кб).

- Аллокация сразу нескольких объектов.

  Вместо аллоцирования одного объекта, можно выделять память сразу для нескольких, что позволяет хранить больше объектов рядом друг с другом. Однако при таком улучшении сложно понимать, в какой момент можно удалять используемый блок.

- Установка верхней границы размера листа со свободными блоками.

  Это поможет бороться с фрагментацией памяти из-за невозможности бесконечного роста.

Интересной особенностью является то, что использование `freelist` возможно с любым другим аллокатором(возможно, при некоторых доработках). Например, при использовании со `stack`-аллокатором появляется возможность работать с блоками внутри последнего.

2.1.6 Affix-аллокатор

Иногда бывает полезным добавить некоторую метаинформацию до и после выделенного блока. Например, можно подписать блок байтами и использовать их в качестве цифровой подписи: если при деаллокации подписи не совпадают, значит что-то пошло не так. Или до и после блока оставить память размером 1Кб и при деаллокации следить, что состояние этих участков не изменилось, иначе был выход за границы памяти [8]. Похожий принцип используется в санитайзерах памяти.

```
template <class Alloc, class Prefix, class Suffix = void>
class AffixAllocator;
```

Такой аллокатор также можно использовать для отлавливания ошибок или дополнительной информации о поведении вашей программы.

Довольно частым приёмом является хранение информации о названии файла и номера строки, в которой был запрос на выделение памяти.

2.1.7 Stats-аллокатор

Довольно полезным также является `stats`-аллокатор [8], который позволяет собирать различную статистику об использовании другого аллокатора: вызовы методов, неудачные операции, кол-во байтов при выделении/удалении, имя файла/номер строки/имя функции/время работы.

```
template <class Alloc, ulong flags>
class StatsAllocator;
```

2.1.8 BitmappedBlock

Такой аллокатор выделяет память блоками равного размера и на каждый блок имеет всего 1 бит метаинформации: занят ли блок [8]. Такая структура лишена большинства недостатков `freelist`.

```
template <class Alloc, size_t blockSize>
class BitmappedBlock;
```

Недостатками такого подхода является фиксированность размера блока, что может привести к фрагментации.

2.1.9 Cascading-аллокатор

Чаще всего `BitmappedBlock` [8] управляет участками памяти внутри большого блока (например множество участков по 64 байта внутри блока размером 1Мб). В случае, когда в какой-то момент 1Мб перестало хватать, необходимо выделить новый большой блок и создать новый `BitmappedBlock` для

него. Этим и занимается `Cascading-аллокатор`. Он лениво создаёт новые участки памяти, пока это требуется, и удаляет неиспользуемые участки. Однако его реализация является не самой тривиальной из-за сложной логики поведения.

```
template <class Creator>
class CascadingAllocator;
auto ca = CascadingAllocator({ return BitmappedBlock<...>(); });
```

2.1.10 Segregator

`Segregator` [8] позволяет применять различные стратегии управления памятью в зависимости от некоторой точки отсчёта:

```
template <size_t threshold,
    class SmallAllocator,
    class LargeAllocator>
struct Segregator;
```

Отметим, что для `segregator` дочерним аллокаторам не нужно реализовывать метод `owns`, потому что размер удаляемой памяти можно сравнить с точкой отсчёта.

Эта композиция очень легко реализуется, что является замечательным дополнением к её мощности.

`Segregator` можно использовать с "самим собой":

```
using MyAlloc = Segregator<4096,
    Segregator<128,
        Freelist<Mallocator, 0, 128>,
        MediumAllocator>,
    Mallocator>;
```

Можно использовать более вложенную структуру и с помощью большого количества таких композиций сделать бинарное дерево поиска в зависимости от размера, что позволяет ускорить поиск нужной стратегии.

2.1.11 Bucketizer

`Bucketizer` [8] позволяет для каждого блока размером `step` в диапазоне размеров от `min` до `max` создавать отдельный аллокатор.

```
template <class Alloc,
    size_t min, size_t max, size_t step>
struct Bucketizer;
```

Часто используются конструкции, позволяющие оперировать блоками, размеры которых растут логарифмически (1, 2, 4, 8, …).

2.1.12 Аллокаторы с разделяемой памятью

Иногда необходимо, чтобы некоторая информация была доступна нескольким процессам без лишних действий вроде сериализации/десериализации данных со стороны пользователя контейнера. В таких случаях используются аллокаторы с разделяемой памятью. Они обеспечивают корректную синхронизацию между процессами для удобного доступа к общим данным.

2.2 Дополнительные главы

2.2.1 Копирование/присваивание аллокаторов

Существуют некоторые проблемы при работе с аллокаторами, возникающие при их копировании/перемещении. Потому разработчик сам должен решить, как будет вести себя аллокатор при этих операциях. Есть два варианта: скопировать объект аллокатора с новым блоком памяти и скопировать в него все значения (теперь имеется два идентичных участка в памяти) или новый аллокатор будет указывать на тот же блок памяти (теперь оба контейнера работают с одним участком).

Для решения такой проблемы в `allocator_traits` существует метод `select_on_container_copy_construction` [11], который возвращает объект аллокатора, используемый в контейнере, в который аллокатор копируется. Перегрузив этот метод, разработчик и может задать нужное поведение.

2.2.2 constexpr allocator

В C++ существует ключевое слово `constexpr`. Оно означает, что при выполнении некоторых условий значение переменной/метода может быть вычислено на этапе компиляции.

В C++17 ещё нет возможности полноценно использовать стандартные контейнеры. Максимум, это контейнеры, выделяющие память на стеке(`std::array`):

```
template <std::size_t N>
constexpr int naiveSumArray() {
    std::array<int, N> arr{0};
    for (size_t i = 0; i < arr.size(); ++i) { arr[i] = i + 1; }
    int sum = 0;
    for (int x : arr) { sum += x; }
    return sum;
}
```

Но при использовании такой функции пользователь обязан передавать размер как параметр шаблона: `naiveSumArray<10>()`.

В C++20 частично решена проблема с динамическими аллокациями. Важным понятием в этой возможности является временная аллокация [12] (англ. transient allocation). Оно означает, что разработчик может динамически выделять память внутри `constexpr`-выражения, но обязан освободить её до окончания этого выражения. При таких ограничениях компилятору гораздо легче реализовать эту возможность и следить за аллокациями. Удобно также то, что в случае, если разработчик забудет освободить память, код просто не скомпилируется.

Перепишем код функции выше с учётом новой возможности:

```
constexpr int naiveSum(unsigned int n) {
    auto p = new int[n];
    std::iota(p, p + n, 1);
    auto tmp = std::accumulate(p, p + n, 0);
    delete[] p;
```

```
    return tmp;
}
```

Теперь можно использовать также и стандартные алгоритмы, которые получили `constexpr`-версии. Код стал гораздо более лаконичным и удобным.

Эта возможность является ключевой для использования `constexpr std::vector` и `constexpr std::string` в будущем [12].

2.2.3 Пропагация аллокаторов

Пропагация аллокаторов – политика переноса функциональности аллокаторов при различных операциях.

Пропагация бывает латеральная или горизонтальная(англ. lateral) и глубокая(англ. deep).

К латеральной пропагации относится всё, что связано с копирующим/перемещающим присваиванием/конструированием и обменом информации. Например, определив в аллокаторе тип `propagate_on_container_copy_assignment` [3], можно указать, что при копировании нужно использовать тот же аллокатор, что и в контейнере, из которого он копируется. По умолчанию аллокатор полностью скопируется.

Глубокая пропагация – приём, который позволяет использовать различные аллокаторы на разных уровнях конструирования сложных объектов.

Рассмотрим следующий код:

```
using String = std::basic_string<char, std::char_traits<char>,
                                 MyAllocator<char>>;
using Vector = std::vector<String, MyAllocator<String>>;
MyAllocator<String> as(some_memory_resource);
MyAllocator<char> ac(as);
Vector v(as); v.push_back(String("hello", ac));
```

Как видим, необходимо явно передавать нестандартный аллокатор для конструирования элементов вектора. Но можно использовать `std::scoped_allocator_adaptor`:

```
using String = std::basic_string<char, std::char_traits<char>,
                                 MyAllocator<char>>;
using Vector = std::vector<String,
    std::scoped_allocator_adaptor<MyAllocator<String>>>;
MyAllocator<String> as(some_memory_resource);
Vector v(as); v.push_back("hello");
```

Код стал значительно проще, ведь теперь контейнер сам передаст используемый аллокатор для конструирования элементов.

2.2.4 pmr::polymorphic_allocator

Устройство

В C++17 наряду с основной моделью аллокатора была введена новая: `std::pmr::polymorphic_allocator` [13] (далее для краткости вместо `std::pmr` будем писать просто `pmr`). `pmr` расшифровывается как Polymorphic Memory Resource, что в целом отражает концепцию работы `pmr::polymorphic_allocator`.

Основной единицей при использовании такого аллокатора является `pmr::memory_resource`(ресурс памяти). Это класс со следующей реализацией:

```
class memory_resource {
public:
  virtual ~memory_resource();
  void* allocate(size_t bytes, size_t alignment);
  void deallocate(void* p, size_t bytes, size_t alignment);
  bool is_equal(const memory_resource& other) const noexcept;
private:
  virtual void*
  do_allocate(size_t bytes, size_t alignment) = 0;
  virtual void
  do_deallocate(void* p, size_t bytes, size_t alignment) = 0;
```

```
  virtual bool
  do_is_equal(const memory_resource& other) const noexcept = 0;
};
```

В случае, когда разработчик хочет реализовать некоторую свою логику работы с памятью, он просто создаёт свой ресурс памяти, который наследует от `pmr::memory_resource`, перегружая методы в приватной части класса. `pmr::polymorphic_allocator` же является просто обёрткой над ресурсом памяти, вызывая у него методы из публичной части класса(которые в свою очередь вызывают виртуальные методы из приватной части).

Вот некоторые стандартные ресурсы памяти:

- `new_delete_resource` - обёртка над `::operator new` и `::operator delete`;
- `null_memory_resource` - при использовании бросает исключение(как `null`-аллокатор).

Три следующих ресурса памяти имеют некоторый родительский ресурс, который они используют в качестве источника памяти в случае, когда кончается память в основном:

- `synchronized_pool_resource` - потокобезопасный ресурс памяти над множеством пулов(pools) одинакового размера;
- `unsynchronized_pool_resource` - непотокобезопасный ресурс памяти над множеством пулов(pools) одинакового размера;
- `monotonic_buffer_resource` - очень быстрый непотокобезопасной неудаляющий ресурс памяти.

Достоинства `pmr::polymorphic_allocator`

`pmr::polymorphic_allocator` решает множество проблем `std::allocator`. Особенно в форме `pmr::polymorphic_allocator<byte>`. Рассмотрим плюсы такого аллокатора:

1. Такой аллокатор подходит абсолютно для любого типа. Ему лишь нужно знать про размер выделяемой памяти и выравнивание. Никакой информации о типе;

2. Глубокая пропагация происходит автоматически: не нужно использования `scoped_allocator_adaptor`. Вместо этого достаточно использовать, например, `pmr::vector` и `pmr::string` вместо стандартных версий;

3. Лучше, чем просто указатель на ресурс памяти, из-за:

   - Корректной инициализации по умолчанию;
   - Не будет переприсваивания аллокатора в случае, если что-то идёт не так;
   - Совместим со стандартной библиотекой.

В силу своих улучшений `pmr::polymorphic_allocator` действительно стал хорошей альтернативой для `std::allocator`.

2.2.5 Некоторые известные аллокаторы

Кратко упомянем популярные способы управления динамической памятью:

- `mmap` [14] – системный вызов в операционных системах семейства UNIX.

  В сообществе принято, что использование `mmap` является самым правильным и стандартным способом выделения памяти для больших размеров (например 20Мб+), однако он является довольно медленным, потому его стараются не использовать для маленьких аллокаций.

- Mimalloc [6] – аллокатор от компании Microsoft;

  Предназначен для небольших маложивущих аллокаций. Разбивает адресное пространство на блоки некоторого размера. Для каждого блока хранит FreeList свободных подотрезков. Для каждого потока память создаётся отдельно. Вкупе с небольшими синхронизациями это делает аллокатор потокобезопасным. Одной из проблем является выбор границы(512Кб),

после которой начинается использование `mmap`. Это слишком замедляет выделение на средних аллокациях. Плюсы: простая реализация, наличие различных политик использования и кеш-дружественность.

- TCMalloc [15] – аллокатор от компании Google;

  TCMalloc имеет несколько уровней кеширования, основным из которых является FreeList, позволяющий аллоцировать блоки по 512 байт, что часто позволяет умещать по несколько объектов в одном участке памяти. Оптимизирован для масштабируемости и использовании в многопоточных приложениях. Позволяет использовать гибкие настройки для получения детальной информации о всех происходящих процессах использования памяти.

- jemalloc [16] – один из вариантов реализации `malloc`;

  Предназначен для выполнения в многопоточных приложениях и решения проблем, связанных с фрагментацией памяти. При сравнении с другими аллокатороми чаще всего показывает наилучшие показатели по потреблению памяти и времени работы.

- dlmalloc [7] – ещё одна из реализаций `malloc`;

  Использует группировку свободных блоков по размерам, объединение граничащих свободных блоков и старается максимизировать кеш-дружественность. Разработан Doug Lea в 1996 году.

- LowFat [17] – аллокатор, используемый для выявления ошибок работы с памятью;

  Является одним из самых быстрых аллокаторов, направленных на нахождение ошибок, связанных с выходом за границы участков памяти. Автором является Gregory J. Duck.

Существует ещё множество других популярных аллокаторов [21]: hoard аллокатор, kmalloc, CAMA, ottomalloc, snmalloc, LRMalloc, lockless, SLAB и т.д.

2.2.6 Аллокаторы в других языках программирования

Несмотря на то, что аллокаторы изначально являются концепцией C++, они имеют место и в некоторых других языках программирования.

В языке программирования Rust:

- Существовало несколько стандартных аллокаторов: сначала это был уже упомянутый jemalloc, после чего он был заменён на свою системную версию, чтобы ослабить зависимость от libc [18].

- Очень легко можно изменить глобальный аллокатор [19]. В то же время в C++ требуется либо постоянно его указывать, либо менять части стандартной библиотеки, что является достаточно нетривиальной задачей.

В языке программирования Zig нельзя выделять память с помощью стандартного глобального аллокатора, т.к. его просто не существует. При необходимости выделить память эффективнее всего будет изучить стандартную документацию [22], где существует набор правил для выбора наиболее подходящего аллокатора из всего множество аллокаторов в этом языке программирования.

Некоторые языки программирования имеют аналогичные/основанные на аллокаторе C/C++ модели управления памятью (например COBOL [23], Fortran [24], D [25] и др.).

ГЛАВА 3

СБОРЩИКИ МУСОРА

3.1 Зачем нужны сборщики мусора

Управление памятью с помощью вызовов различных методов для выделения/ удаления памяти и аллокаторов позволяет добиваться высокой производительности, однако не является наиболее удобным с точки зрения использования. При реализации аллокатора всё так же можно забыть освобождать неиспользуемую память, что приведёт к масштабным утечкам памяти во всей программе. Для достижения высокой эффективности необходимо разбираться в операционных системах, модели памяти, низкоуровневом программировании, что автоматически делает такие инструменты сложноосваиваемыми, недоступными для начинающих программистов.

Потому существует другая фундаментальная концепция управления памятью, называемая сборкой мусора: специальный процесс, называемый сборщиком мусора (англ. garbage collector), периодически очищает память от ставших ненужными объектов. Реализация корректного сборщика мусора является довольно нетривиальной задачей, ведь она включает в себя сложный анализ жизненного цикла объектов (удаление без утечек памяти и отслеживание всех использований конкретного объекта), что приводит к меньшей эффективности сборщиков мусора перед ручным управлением памятью. Чаще всего этим занимаются большие команды для нужд собственной компании и сообщества, однако это облегчает жизнь множеству программистов, использующих языки программирования с автоматическим управлением памятью.

3.2 Алгоритмы сборки мусора

Методы сборки мусора чаще всего классифицируются на две категории: трассирующая сборка мусора и прямая.

Трассирующая сборка мусора – сборка мусора, при которой осуществляется отслеживание достижимых объектов и время от времени запускается некоторый процесс, который удаляет ненужные объекты, переносит память из одной области в другую и т.д.

Прямые методы сборки мусора – это методы, которые не откладывают решение об (не)удалении объекта на потом, а разбираются с этим в момент каких-то изменений объекта.

Также некоторые методы передвигают объекты в памяти (для борьбы с фрагментацией), а некоторые нет.

3.2.1 Mark-sweep

Mark-sweep - один из самых тривиальных трассирующих алгоритмов сборки мусора.

Он делится на два этапа: mark и sweep.

Каждый объект имеет свой бит достижимости (0 - не достижим, 1 - достижим). Изначально все объекты кроме корневого (некоторый специальный системный объект, который всегда достижим) считаются недостижимыми. В некоторый момент исполнение программы останавливается (подобное поведение обозначается термином stop-the-world) и начинается фаза mark: поиск в глубину из корневых объектов, во время которого у всех достижимых объектах бит достижимости устанавливается в значение 1. После обхода всех объектов начинается фаза sweep: обходятся абсолютно все объекты. В случае, если бит достижимости установлен в 1, то он просто сбрасывается. Если же он равен 0, то объект считается мусором и с ним происходят необходимые манипуляции: чаще всего это добавление указателя в freelist (оригинальный mark-sweep) либо перегруппировка объектов в памяти так, чтобы уменьшить фрагментацию памяти (вариация под названием mark-compact). Очевидными недостатками такого подхода являются блокировка всей программы на некоторое время и обход

всей области памяти программы.

Существует улучшенная версия этого алгоритма под названием BF-Mark, которая избавлена от этих недостатков. Вместо двух «цветов» достижим/нет объект используется 3: множество чёрных объектов, множество серых и множество белых. Чёрные объекты – объекты, доступные из корней и не имеющие исходящих ссылок на белые объекты, белые – кандидаты на удаление, серые – объекты доступные из корней, но пока не проверенные на наличие ссылок на белые объекты. Алгоритм состоит из трёх шагов: выбрать объект из серого множества и переместить его в чёрное; поместить все белые объекты, на которые есть ссылки из нового чёрного в серое множество; повторять прошлые два шага, пока серое множество не станет пустым. Когда серый набор пуст, сканирование завершено: черные объекты доступны из корней, в то время как белые объекты недоступны и могут быть собраны мусором.

Поскольку все объекты, до которых невозможно добраться сразу из корней, добавляются к белому набору, а объекты могут перемещаться только от белого к серому и от серого к черному, алгоритм сохраняет важный инвариант - никакие черные объекты не ссылаются на белые объекты. Это гарантирует, что белые объекты могут быть освобождены после того, как серый набор станет пустым.

Трехцветный метод имеет важное преимущество - его можно выполнять «на лету», не останавливая систему на значительный период времени: каждый удалённый/выделенный объект помещается в серое множество, после чего «дообходится». Независимо от того, достижим ли объект фактически, он считается достижимым и удалится на следующей итерации сборки мусора, т.к. лучше не убрать лишний мусор, чем убрать что-то необходимое.

3.2.2 Копирующий сборщик

Кроме mark-sweep довольно известным методом очистки памяти является копирующий сборщик [33] (иногда его называют semispace, Lisp 2 или алгоритмом Чейни).

Алгоритм основывается на том, что выделяется две области памяти одинакового размера. Все объекты создаются в одной из них, вторая при этом содержится пустой. Как и в прошлом алгоритме, есть несколько корневых объектов,

которые ссылаются на другие. В некоторый момент все объекты проверяются на достижимость. В случае, если объект валиден, он дублируется во вторую пустую область памяти, иначе удаляется. В итоге все достижимые объекты скопированы в новое место. Произошла очистка ненужных объектов и уплотнение для избежания фрагментации.

3.2.3 Подсчёт ссылок

Одним из самых известных нетрассирующих сборщиков мусора является считающий ссылки сборщик мусора. Проверка достижимости происходит очень просто: в случае, когда счётчик ссылок у объекта равен нулю, его можно удалять. Однако такая тривиальная реализация не учитывает компоненты связности объектов, где вся группа недоступна, но объекты указывают друг на друга. В таком случае можно усовершенствовать алгоритм и производить поиск циклов или их конденсацию в графе объектов, чтобы бороться с подобными ситуациями. Такой сборщик мусора является довольно простым в реализации, однако имеет значительный недостаток – все операции со ссылками становятся чуть медленнее, в силу того, что до и после всех операций, которые могут изменить количество ссылок на объект, приходится это учитывать. Эти участки кода называются барьерами сборщика мусора (англ. gc bariers). Барьеры также используются в многопоточном BF-Mark.

3.2.4 Сборка поколениями

Одним из самых эффективных алгоритмов сборки мусора является сборщик мусора с поколениями [32]. Он основан на слабой гипотезе о поколениях: «Большинство объектов умирают молодыми». Гипотеза называется слабой, т.к. они не обязана выполняться: всегда можно написать приложение, для которого это утверждение неверно. Однако гипотеза описывает некоторый средний случай, для которого сборка поколениями является довольно эффективной.

Новые объекты считаются объектами первого поколения. Если эти объекты переживают n сборок мусора, их поколение становится вторым. Объекты более высокого поколения проверяются реже, т.к. подразумевается, что раз они уже долго прожили, то и далее проживут дольше «молодых» объектов. В случае,

если они опять переживают несколько сборок мусора своего поколения, то они перемещаются в третье поколение. Количество поколений обычно ограничено каким-то небольшим числом (т.е. не растут бесконечно).

Правило при обходе графа объектов простое — если нам повстречался объект из более старшего поколения, чем проверяемое в данный момент, дальше в этом направлении не идем. Однако здесь есть тонкий момент. Поскольку объекты в общем случае могут изменяться, они также могут содержать ссылки на объекты более молодого поколения. Поэтому во время выполнения программы необходимо отслеживать ситуации, когда более старый объект начинает ссылаться на более молодой и добавлять в этом случае молодой объект в «корневое множество» объектов соответствующего поколения. Иначе сборщик мусора ошибочно удалит его, как недостижимый.

Иногда полезно знать, как устроен сборщик мусора, которым пользуется разработчик, чтобы не бояться создавать этот самый мусор. Например объекты можно переиспользовать, однако для сборщика мусора с поколениями это будет означать, что объект долгоживущий, из-за чего его поколение вырастет и он будет проверяться реже. Это искуственное продлевание жизни объекта в итоге может сделать операции сборщика мусора менее эффективными.

3.3 Характеристики сборщиков мусора

В сообществе принято выделять 3 основных характеристики сборщиков мусора:

- Пропускная способность - количество операций в единицу времени;
- Предсказуемость времени остановки приложения;
- Объём используемой памяти.

Подобно CAP-теореме невозможно оптимизировать одновременно все показатели, потому приходится от одного отказываться и находить подходящую композицию из двух других.

Также имеется поведенческая характеристика сборщиков мусора: являются ли они stop-the-world. Такие сборщики мусора на время останавливают выпол-

нение приложения и очищают все ненужные объекты. Такая характеристика может показаться заведомо отрицательной, однако подобные сборщики мусора пишутся гораздо легче и являются хорошим решением для некритичных сервисов. Рассмотрим некоторую периодическую задачу, которая выполняется 2 часа. Для неё неважно, будет суммарная остановка 10 ил 15 секунд: это всё равно очень мало. В то время как в постоянно нагруженных сервисах приостановка работы одного сервера на такой промежуток может привести к повышению нагрузки на других, что может оказаться фатальным.

Как альтернатива полностью stop-the-world сборщикам мусора есть два метода, уменьшающих паузы: фоновая и инкрементальная сборки. Фоновая сборка мусора осуществляется в отдельном потоке, что позволяет не останавливать приложение на большие промежутки (затраты на сборку мусора распределены на всё выполнение программы). Инкрементальная сборка так же останавливает выполнение приложения, однако это делается много раз на небольшие отрезки времени, что позволяет не замечать задержки. Чаще всего суммарные затраты на инкрементальную/фоновую сборку мусора гораздо больше, чем простой stop-the-world, в силу более сложных методов и поддерживаемых структур.

3.4 Сборщики мусора в различных языках программирования

Существует множество различных языков программирования, использующих автоматическую сборку мусора. Некоторые из них имеют какие-то простые сборщики мусора, а некоторые — сложные композиции различных стратегий. В некоторых сборщики мусора по умолчанию стараются делать эффективными и подходящими для различных ситуаций, а где-то разработчикам на выбор даётся несколько сборщиков со своими плюсами и минусами и множество возможностей их настроить.

3.4.1 Java

Язык программирования Java является имеет одну из самых сложных и интересных систем сборок мусора (точнее несколько).

EpsilonGC

Необычным сборщиком мусора является EpsilonGC [37]. Он старается осуществлять максимально быстрые аллокации для короткоживущих приложений. Достигается это с помощью пустого этапа сборки мусора, что позволяет не поддерживать никаких сложных структур, не тратить время на сихронизации, не делать stop-the-world приложения. Выделение памяти аналогично pooled-аллокаторам.

G1

В Hotspot JVM существует несколько популярных сборщиков мусора [34]: SerialGC, ParallelGC, CMS и G1. Рассмотрим каждый из упомянутых сборщиков мусора.

SerialGC — последовательный сборщик мусора, состоящий из DefNEW и Tenured. Первый занимается сборкой молодого поколения и является копирующим сборщиком мусора, второй же следит за взрослым поколением и является модифицированным алгоритмом mark-sweep-compact. В силу того, что на обоих этапах происходит перемещение, аллокация аналогично pooled-аллокаторам.

ParallelGC (ParNEW) — параллельный сборщик мусора, состоящий из параллельного копирующего для молодого поколения и параллельного mark-compact для взрослого поколения. Аллокация также линейная. Несмотря на параллельность, он всё ещё является stop-the-world, однако паузы в среднем меньше.

CMS — параллельный сборщик мусора, пытающийся минимизировать паузы приложения и использовать фоновую сборку. При первой остановке приложения фиксируется граф объектов. Далее в фоновом режиме этот граф обходится. На следующей остановке учитываются изменения, которые могли произойти за время фонового обхода, после чего в фоне происходит sweep этап. Структурно используется параллельный копирующий сборщик для молодого поколения, а для взрослого поколения фоновый mark-sweep. Аллокация происходит из freelist’ов, что приводит к фрагментации. Компактификация происходит только в случае, когда памяти начинает не хватать, что приводит к полной

сборке мусора (для CMS это полный stop-the-world в один поток).

G1 (garbage-first) является сборщиком мусора общего назначения и основным в Hotspot JVM. Он является параллельным фоновым сборщиком мусора, который пытается минимизировать паузы приложения. Основной абстракцией является сборка поколениями: существует молодое и взрослое поколение. Молодое поколение содержит три участка памяти (более устоявшимся является термин пул от англ. pool): основной (eden) и два дополнительных. Основной пул содержит самые молодые объекты, два дополнительных используются для копирующей сборки мусора внутри молодого поколения (то есть внутри молодого поколения существует разделение на подпоколения). Когда в молодом поколении недостаточно места для новой аллокации, происходит малая сборка мусора. Она заключается в том, что все выжившие в eden объекты перемещаются в один из пулов для копирующей сборки. Элементы из копирующего сборщика могут быть либо удалены, либо перемещены внутри копирующего сборщика, либо перенесены во взрослое поколение (если сборщик мусора примет такое решение):

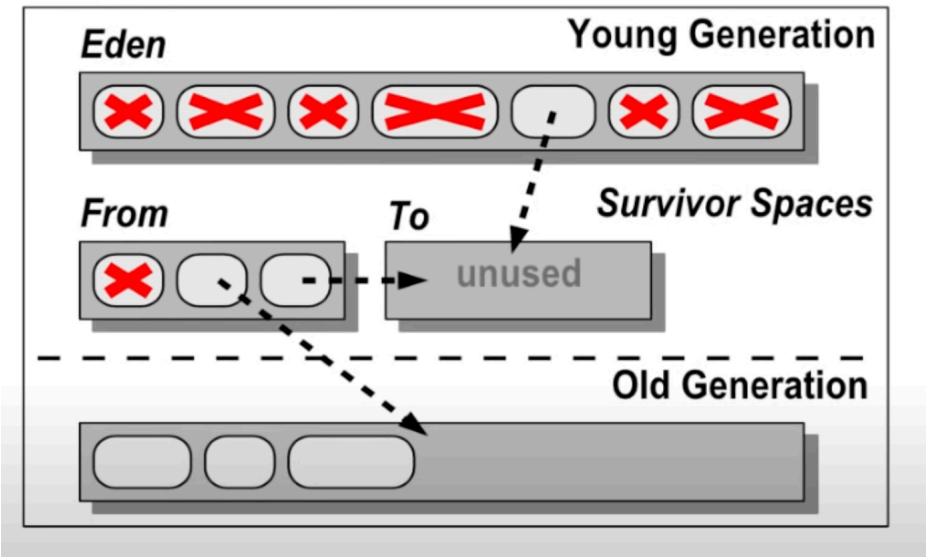

Рисунок 3.1 — Упрощённая схема G1

Вся куча приложения делится на регионы размером от 1Мб до 32Мб. Каждый регион динамически относится в молодое/взрослое поколение (после сборок мусора регионы могут возвращаться во множество свободных и получать новую роль по необходимости). В случае молодого регион может выполнять роль eden или survivor. Особенная роль отводится большим объектам, которые

не могут поместиться в один регион. Для них выделяется несколько рядом лежащих регионов. В процессе сборки мусора для таких объектов применяется отдельная логика.

Во время сборки мусора выбирается множество регионов, которые будут очищаться (collection set). В него входят все регионы из молодого поколения и возможно некоторые из взрослого поколения (тут есть разделение по типу collection set'a: сборка только в молодых регионах, смешанная сборка и полная сборка). Недостижимые объекты удаляются, а достижимые перемещаются в свободные регионы, которые назначаются либо регионами для взрослого поколения, либо survivor-регионами. При хорошем выборе регионов для очистки и засчёт компактификации новые регионы занимают меньше места, чем занимали объекты ранее (но не обязательно).

Для того, чтобы понимать, какие регионы из взрослого поколения стоит брать в очередной collection set, у каждого региона имеется remembered set, хранящий взаимосвязи объектов между регионами.

Во время этапа фоновой маркировки (все остальные этапы являются stop-the-world), которая стартует при достижении некоторого порога заполненности всей памяти приложения, происходит несколько действий: обновляется информация о достижимости по регионам, освобождение регионов без живых объектов, устранение циклических зависимостей между неживыми объектами.

Shenandoah

Другим популярным сборщиком мусора является Shenandoah [35][36].

Если проводить сравнить принцип действия рассмотренных сборщиков мусора, то схематично он будет выглядеть так:

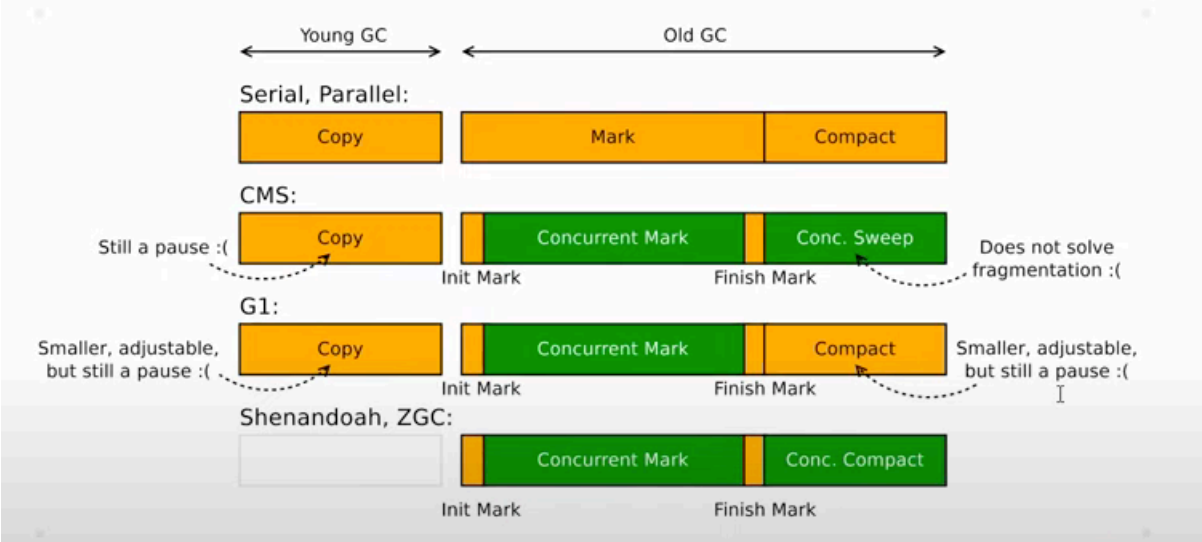

Рисунок 3.2 — Сравнительная схема сборщиков мусора в Java

На текущий момент Shenandoah не использует популярную технику с поколениями, что позволяет не хранить метаинформацию вроде remembered set.

Рассмотрим цикл работы Shenandoah.

Так же, как и у G1, вся память приложения делится на регионы. Первым этапом является маркировка, которая происходит в три фазы:

1. Инициализирующая фаза маркировки, которая является stop-the-world.
2. Параллельная маркировка.
3. Завершающая пауза (также stop-the-world).

В результате для каждого занятого региона становится понятно, как много живых объектов в нём содержится. Из регионов с небольшим количеством достижимых объектов формируется collection set, из которого происходит фоновое уплотнение в новый регион (concurrent evacuation). После уплотнения нельзя считать, что регионы, из которых происходило уплотнение, можно ощищать. На данные в них могут существовать ссылки из других регионов, потому следующим этапом производится перезапись ссылок с данных из старых регионов на данные в уплотнённом (concurrent update refs). Эта часть работы сборщика тоже обрамлена паузами до и после. После этого можно считать регионы очищенными и аллоцировать из них память.

Как и в G1, для Shenandoah полная сборка мусора (FullGC) является наихудшим случаем, который, тем не менее, сумеет разобраться с абсолютно любой

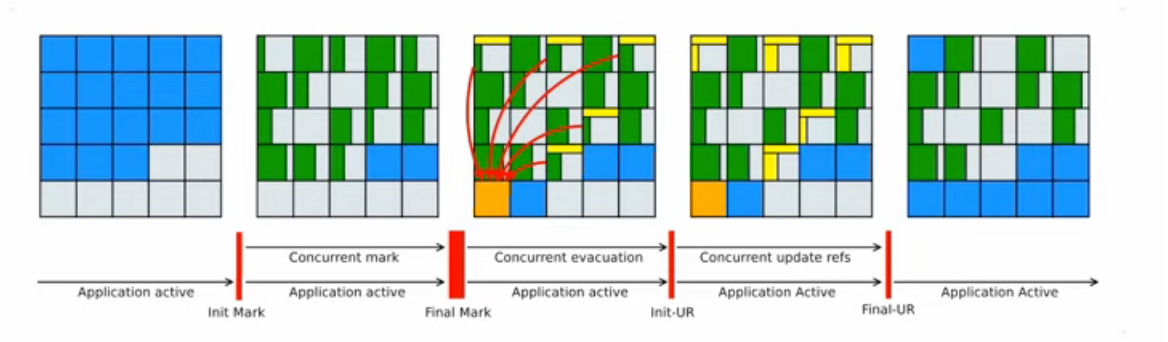

Рисунок 3.3 — Цикл работы Shenandoah

ситуацией, при этом ничего не гарантируя про время этапа stop-the-world.

Ещё одним известным сборщиком мусора в Java является ZGC [41]. По принципу действия он аналогичен Shenandoah с точностью до некоторых локальных оптимизаций, которые, тем не менее, иногда играют значительную роль в использовании. Важным достижением ZGC являются константные stop-the-world паузы.

3.4.2 Python

Стандартный CPython использует подсчёт ссылок и сборщик мусора с поколениями [39]. Второй необходим в силу того, что стандартный алгоритм подсчёта ссылок не учитывает циклы, возникающие в процессе жизнедеятельности программы.

Сборщик мусора имеет три поколения, для которых можно задавать границы того, сколько сборок мусора должен пережить объект, чтобы переместиться в следующее поколение. Для разрешения циклов используется факт, что циклы могут возникнуть лишь при использовании различных контейнеров. Для каждого объекта внутри контейнера производится следующая операция: для всех объектов внутри этого контейнера, на которые ссылается зафиксированный объект, количество ссылок уменьшается на 1. Для всех объектов, счётчик ссылок которых остался больше 1, считается, что на них ссылаются объекты снаружи контейнера. Эти объекты удаляем из множества кандидатов на удаление. Также удаляем все объекты, на которые ссылаются только что удалённые и т.д. Оставшиеся объекты можно удалять.

Подобный подход встречается и в других языках. Например, в Kotlin/Native

такой выбор был сделан из-за простоты метода. Однако недавно от такого подхода решили отказаться в силу того, что с ним не получается достигнуть достаточной эффективности.

В отличие от подсчёта ссылок, сборщик мусора можно отключать. Это может быть полезно для эффективности приложения в случае, когда автор уверен, что приложение не создаёт циклов, или же готов на некоторые утечки памяти в угоду скорости. То есть получается некоторый аналог неудаляющего аллокатора.

Другой интерпретатор python PyPy имеет сборщик мусора с другой моделью поведения [40]. Incminimark — инкрементальный трассирующий сборщик мусора с двумя поколениями. Основная настройка — размер молодого поколения (nursery). Большие объекты создаётся вне поколений.

И CPython, и PyPy предоставляют множество возможностей для полуавтоматического управления сборкой мусора. Есть возможности вручную запускать отдельные минорные/мажорные сборки, временно включать/отключать сборщик мусора, получать подробную информацию о поведении программы при сборке и т.д. Это позволяет эффективнее анализировать происходящее и настраивать сборщик для своих конкретных нужд.

3.4.3 Golang

В языке программирования Go используется улучшенная версия BF Mark, которая неплохо работает в многопоточной среде. Интересным достижением сборщика мусора в этом языке является то, что этап stop-the-world занимает константное время, как у ZGC в Java. Единственная возможность настройки сборщика мусора – это переменная окружения GOGC [31]. GOGC есть размер всей кучи приложения относительно общего размера всех достижимых объектов, т.е. GOGC равное 100 означает, что размер всей кучи приложения на 100% больше размера всех достижимых объектов. Если требуется уменьшить общее количество времени работы сборщика мусора, стоит увеличить GOGC. Если же допускается отдать больше времени сборке мусора, но выиграть себе память — нужно уменьшить GOGC.

3.4.4 JavaScript

В одном из самых популярных компиляторов для языка программирования JavaScript под названием V8 используется композиция нескольких сборщиков мусора [32]. Основной абстракцией при сборке мусора является сборка мусора с поколениями. Тут есть 3 поколения, причём каждое лежит в своём сборщике мусора. Самые молодые объекты появляются в оптимизированном mark-compact сборщике мусора. Обычно он добавляет освободившиеся блоки во freelist, но иногда принимает решение запустить операцию уплотнения. Следующее поколение – объекты «подростки», которые содержатся в копирующем сборщике мусора. Последнее поколение живёт в простом mark-sweep сборщике. Из-за того, что он чистится довольно редко, высокая эффективность тут не требуется.

3.4.5 Другие языки

После рассмотрения сборщиков мусора в различных языках программирования становится понятно, что все методы в итоге сводятся к одному из базовых методов или их комбинации.

В C# используется оптимизированный mark-compact с тремя поколениями [42]. В D используется сборщик мусора со stop-the-world этапами маркировки и фоновым sweep этапом [44]. В языке программирования Nim существует несколько сборщиков мусора [43]: подсчёт ссылок; mark-sweep; boehm; сборщик мусора Golang; оптимизированный подсчёт ссылок; оптимизированный подсчёт ссылок с разрешением циклов; отсутствие сборщика мусора. Также сборщики мусора используются в других известных языках программирования [45]: BASIC, Dylan, Lisp и lisp-подобные языки, Lua, ML, Prolog и др.

Становится понятно, что подходов и правда различное множество, пусть они и строятся на нескольких базовых методах, композиция которых в самых разных формах позволяет создавать многообразие сборщиков. Также видим, что не существует универсального сборщика мусора: каждый язык программирования, преследуя какие-то свои цели, строит систему, подходящую именно ему; более того, в рамках одного языка иногда хотят решать разные задачи, потому создаются сторонние решения.

ГЛАВА 4

РАЗРАБОТКА СБОРЩИКА МУСОРА ДЛЯ C++

Несмотря на то, что в C++ основным и по факту единственным способом управления памятью являются ручные методы и идиома RAII, были попытки сделать условия для появления сборщика мусора в C++11 [46], которые, тем не менее, успехом не увенчались и спустя 12 лет эта часть стандарта была удалена [30].

4.1 Проектирование

Одной из наиважнейших проблем при использовании различных методов управления памятью является фрагментация. Она возникает после большого числа операций выделения/удаления памяти, когда среди занятых областей появляется множество свободных отрезков памяти, которых суммарно достаточно для выделения нового объекта, но каждый из них в отдельности недостаточного размера, что не позволяет найти область для выделения этого объекта.

Для борьбы с этой проблемой реализуем mark-compact сборщик мусора. Будем хранить не два буфера конкретного размера, а несколько блоков фиксированного размера в односвязном списке, что позволит при нехватке памяти выделять новый блок и отдавать память в нём. В таком подходе не теряется преимущество выделения памяти из буфера: если в блоке ещё достаточно памяти, просто отдаём указатель на начало свободного отрезка, т.е. не тратим время не выделение нового блока памяти из операционной системы.

Также важной задачей является достижимость объектов. Особое значение при её решении отводится корневым объектам. Чаще всего корневыми объектами являются различные статические переменные и переменные на стеке. К сожалению, C++ не имеет инструментов вроде рефлексии для решения подоб-

ных задач, однако можно проэмулировать поведение собственноручно, обязав пользователя к примеру наследовать свои объекты от некоторой структуры.

4.2 Реализация сборщика мусора

Разберём реализацию спроектированного сборщика мусора на C++.

Сборщик мусора оформлен как класс-одиночка [29]. Это позволяет использовать один и тот же его экземпляр в любой точке программы.

Так как для каждого объекта необходимо хранить некоторую метаинформацию, а делать это автоматически для любого типа язык не позволяет, обязуем пользователя в своём типе иметь следующую структуру (для случаев, если если этот тип указывает на другие объекты, которые необходимо уметь удалять):

```
enum {
  kStructSz = 0,
  kRefCount = 1,
};
struct Header {
  union {
    int meta[2];
    Header* new_addr;
  };
};
```

В силу особенностей, которые будут использоваться далее, необходимо, чтобы все ссылки находились сразу после данной структуры. В `meta[kRefCount]` будет храниться количество объектов, на которые указывает рассматриваемый объект. В `meta[kStructSz]` будет храниться размер структуры в байтах. Назначение указателя `new_addr` будет рассмотрено ниже.

Использование описанной выше структуры позволит обходить объекты, работающие со сборщиком мусора. Однако при обходе любого графа необходимо иметь начальную вершину. Т.к. классическим решением при выборе корней являются переменные на стеке, то их и выберем. Однако проходясь по стеку

невозможно по байтам понять, что за данные на нём хранятся. Потому будем записывать указатели явно:

```
static std::vector<Header**> stack_refs;
template<class T>
struct Ref {
  T* ref = nullptr;
  Ref() {
    stack_refs.push_back(reinterpret_cast<Header**>(&ref));
  }
  ~Ref() { stack_refs.pop_back(); }
};
```

Принцип обёртки данной функции прост: при создании адрес объекта сохраняется в стек ссылок, а при разрушении — удаляется.

Предварительные приготовления окончены. Далее необходимо научиться выделять память.

В силу выбранного способа работы с памятью (односвязный список блоков) выделять память для нового объекта очень просто: при необходимости выделить новый блок, выдать внутри текущего блока указатель и сдвинуть его на размер структуры.

Размер одного блока выберем в 2048 байт:

```
constexpr static int kChunkSize = 2048;
struct Chunk {
  Chunk* next;
  char data[kChunkSize];
};

static chunk* list_begin;
static chunk* cur_chunk;
static int list_sz;
static int offset;
```

Для реализации односвязаного списка храним начало списка и последний выделенный блок, а также количество выделенных блоков. `offset` отвечает

за смещение внутри последнего блока для быстрого выделения памяти. `char data[kChunkSize]` — буфер, из которого в текущем блоке будет выделяться память.

Рассмотрим выделение памяти:

```
static Header* BufAllocate(int size, int ref_cnt) {
  if (size > kChunkSize) { return nullptr; }
  if (offset + size > kChunkSize) {
    ++list_sz;
    cur_chunk->next = new Chunk();
    cur_chunk = cur_chunk->next;
    cur_chunk->next = nullptr;
    offset = 0;
  }
  auto new_obj = reinterpret_cast<Header*>(
                    &(cur_chunk->data[offset]));
  new_obj->meta[kStructSz] = size;
  new_obj->meta[kRefCount] = (ref_cnt << 1) | 1;
  offset += size;
  constexpr static int kAlign = 8;
  if (offset % kAlign != 0) {
    offset += kAlign - offset % kAlign;
  }
  return new_obj;
}
```

При выделении памяти считаем, что объект выделяемого типа имеет ранее описанную структуру со служебной информацией, чего достаточно для выделения. Также на всякий случай сделаем выравнивание на 8 байт, хотя почти наверняка компилятор осуществит это сам.

Рассмотрим следующую строчку:

```
new_obj->meta[kRefCount] = (ref_cnt << 1) | 1;
```

В силу устройства механизма `union` в C++ неизвестно, какой из членов `union` в данный момент в нём находится. Зная, что указатель `new_addr` выравнен на

8 байт, можем понять, что его три младших бита всегда заполнены нулями. Потому при записи количества ссылок установим один бит в единицу, что позже поможет понять: хранится тут указатель или массив из двух чисел. Потому будет легко понять, что мы храним:

```
static bool IsPointer(Header header) {
  return (header.meta[kRefCount] & 1) == 0;
}
```

И небольшая обёртка для более удобной работы:

```
template <typename T, typename... Args>
static T* Alloc(Args&&... args) {
  auto ptr = reinterpret_cast<T*>(
            BufAllocate(sizeof(T), T::kRefCnt));
  if constexpr (sizeof...(args) > 0) {
    new (ptr) T(std::forward<Args>(args)...);
  }
  return ptr;
}
```

Тут заметим, что тип должен предоставлять статическую переменную `kRefCnt`, которая равна количеству объектов, с которыми должен уметь работать разработанный сборщик мусора.

Перед использованием необходимо сделать инициализацию сборщика мусора:

```
static void Init() {
  list_begin = cur_chunk = new Chunk;
  list_begin->next = nullptr;
  offset = 0;
  list_sz = 1;
}
```

Рассмотрим процесс сборки мусора. Он происходит при вызове метода `Collect`:

```
static void Collect() {
  auto new_first_chunk = cur_chunk = new Chunk;
  cur_chunk->next = nullptr;
  offset = 0;
  list_sz = 1;
  for (auto ref: stack_refs) { Move(ref); }
  auto iter = list_begin;
  list_begin = new_first_chunk;
  while (iter) {
    auto t = iter->next;
    delete[] iter;
    iter = t;
  }
}
```

Для начала необходимо создать новую область память для перемещения в неё достижимых объектов. Далее обходим все корни (указатели на стеке). От каждого корня идём поиском в глубину и все достижимые объекты перемещаем в новую область памяти. И в конце освобождаем старую область памяти, в которой остались недостижимые объекты.

Стоит обратить внимание, что у удаляемых объектов не вызываются деструкторы. Это не проблема для классов, которые в них лишь очищают память, но может стать проблемой для случаев, когда в деструкторе закрывается некоторые соединение. Потом часто в языках программирования со сборкой мусора подобные действия нужно делать явно.

Осталось разобраться с функцией `Move`:

```
static void Move(Header** current) {
  if (*current == nullptr) { return; }
  if (IsPointer(**current)) {
    (*current) = (*current)->new_addr;
    return;
  }
  auto new_obj = BufAllocate((*current)->meta[kStructSz],
                             (*current)->meta[kRefCount]);
  std::memcpy(new_obj, (*current),
        sizeof(char) * (*current)->meta[kStructSz]);
  (*current)->new_addr = new_obj;
  (*current) = new_obj;

  auto next_ref = reinterpret_cast<Header**>(new_obj) + 1;
  int ref_cnt = new_obj->meta[kRefCount] >> 1;
  for (int i = 0; i < ref_cnt; ++i, ++next_ref) {
    Move(next_ref);
  }
}
```

Для начала выделим новую область памяти и скопируем всё содержимое объекта. В поле `new_addr` запишем новый адрес объекта. В случае, если на этот объект существуют другие ссылки, в `Header` будет хранится указатель на новую область памяти, где этот объект хранится, потому просто перенаправим указатель на эту область памяти. Далее используем тот факт, что ссылки лежат в начале объекта: двигаемся по каждой из них и осуществляем перемещение объекта, на который эта ссылка указывает.

На этом реализация сброщика мусора окончена. Для удобства использования допишем несколько строк кода — структура, от которой нужно наследоваться и несколько макросов:

```
template<int RefCnt>
struct ObjHeader {
```

```
  Header gc;
  static constexpr int kRefCnt = RefCnt;
};

#define DECL(type) GC::Ref<type>
#define NEW(type) GC::Alloc<type>()
#define NEW_CONSTRUCT(type, ...) GC::Alloc<type>(__VA_ARGS__)
```

Тестирование осуществлялось на простом бинарном дереве поиска при помощи ”отрезания”поддерева от родительской вершины, после чего наблюдалось очищение всех вершин этого поддерева. Код теста и его результаты можно найти в Приложении 1.

4.3 Достоинства и недостатки разработанного сборщика мусора

Самым большим достоинством разработанного решения является то, что это сборщик мусора для языка программирования C++, который работает.

Рассмотрим недостатки:

- Решение не подходит для объектов, имеющих виртуальных методы. Несмотря на то, что это решаемая проблема, она требует специфических знаний о языке программирования C++, что лишь усложнит код, что неудобно, если мы хотим разработать прототип сборщика мусора.

- Обязуем пользователя наследоваться от некоторой структуры для хранения служебной информации и располагать ссылки на объекты в самом начале структуры. Расположение ссылок в объекте решается вместе с решением проблемы использования сборщика для объектов с виртуальными методами. Проблему наследования к сожалению никак не решить, т.к. C++ просто не предоставляет никаких инструментов вроде рефлексии для решения необходимых задач. Учитывая это, также понятно, невозможно использовать текущую реализацию со стандартной библиотекой, т.к. она не выполняет наши требования.

- В текущей версии невозможно выделение объектов размером больше размера одного блока. Это проблема легко решается, однако код лишь усложнился бы без наличия полезной идеи.

- Вызывать сборку мусора нужно самому. Однако тут это скорее идеологическое решение, ведь в C++ считается хорошей практикой делать поток исполнения прозрачным, потому поведение сборщика мусора из любого другого языка может считаться плохо спроектированным решением.

4.4 Направления для развития

После разработки прототипа работающего решения можно развивать его в нескольких направлениях. Рассмотрим некоторые из них:

- С точки зрения концептуальной составляющей можно усложнить подобный сборщик мусора, добавив несколько поколений, в которых можно использовать другие стратегии аналогично другим языкам.

- В данной реализации блоки перемещаются согласно обходу в глубину, однако лучше всего было бы перемещать объекты согласно статистике обращений, чтобы повысить локальность данных.

- Вместо простого выделения памяти блоками можно хранить например freelist с блоками некоторого размера аналогично аллокаторам, чтобы повысить эффективность.

- Параметризировать различные параметры сборщика мусора в начале работы.

- Добавить поддержку многопоточности. К сожалению это один из самых сложных пунктов. Т.к. ни один из сборщиков мусора не может работать без stop-the-world этапа, необходимо останавливать выполнение всех потоков, что является очень сложной задачей, т.к. придётся написать около половины компилятора C++.

ЗАКЛЮЧЕНИЕ

Сборка мусора – очень важный инструмент в большинстве языков программирования. Чаще всего это довольно нетривиальные методы, от которых напрямую зависит эффективность разрабатываемого приложения. Разработать быстрый, не требовательный к памяти сборщик мусора — сложная техническая задача. Сделать это для языка программирования C++ на языке программирования C++ — задача ещё сложнее. После изучения различных методов управления памятью в C++ и подходов к сборке мусора (как фундаментальных, так и на конкретных примерах) удалось реализовать сборщик мусора для C++.

С одной стороны разработанное решение справляется с поставленной задачей. С другой — обладает некоторым количеством недостатков, которые невозможно или очень сложно решить в рамках даже дипломной работы. Подобные проблемы должны исправляться на уровне спецификации языка программирования и поддерживающих её компиляторов, т.к. требуют решение проблем в самых различных частях C++. При анализе существующих проблем было выяснено, что для C++, как и для других языков программирования, полноценный сборщик мусора может существовать лишь при плотной интеграции в стандартную библиотеку, а лучше при полной интеграции его в язык. Однако подобные решения неизбежно приведут к снижению эффективности языка, который позиционируется как один из самых лучших инструментов, когда речь идёт о скорости выполнения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Презентация модели стандартной библиотеки комитету по развитию C++ от Александра Степанова. [Электронный ресурс] / - Режим доступа: http://stepanovpapers.com/Stepanov-The_Standard_Template_Library-1994.pdf. - Дата доступа: 05.11.2021.

2. Официальная документация по языку программирования C++. `std::allocator` [Электронный ресурс] / - cppreference.com - Режим доступа: https://en.cppreference.com/w/cpp/memory/allocator. - Дата доступа: 06.11.2021.

3. Официальная документация по языку программирования C++. `std::allocator_traits` [Электронный ресурс] / - cppreference.com - Режим доступа: https://en.cppreference.com/w/cpp/memory/allocator_traits. - Дата доступа: 08.11.2021.

4. LLVM - реализация стандартной библиотеки C++. [Электронный ресурс] / - github.com/llvm-mirror - Режим доступа: https://github.com/llvm-mirror/libcxx/blob/master/include/memoryL1465. - Дата доступа: 12.11.2021.

5. Официальная документация по языку программирования C++. `std::vector` [Электронный ресурс] / - cppreference.com - Режим доступа: https://en.cppreference.com/w/cpp/container/vector. - Дата доступа: 14.11.2021.

6. Аллокатор Mimalloc. [Электронный ресурс] / - github.com/microsoft - Режим доступа: https://github.com/microsoft/mimalloc. - Дата доступа: 15.11.2021.

7. Аллокатор dlmalloc. [Электронный ресурс] / - gee.cs.oswego.edu - Режим доступа: http://gee.cs.oswego.edu/dl/html/malloc.html. - Дата доступа: 19.11.2021.

8. Конференция CppCon 2015. [Электронный ресурс] / - std::allocator... Andrei Alexandrescu. - Режим доступа: https://youtu.be/LIb3L4vKZ7U. - Дата доступа: 23.11.2021.

9. Официальная документация по языку программирования C++. `std::pmr::monotonic_buffer_resource` [Электронный ресурс] / - cppreference.com - Режим доступа: https://en.cppreference.com/w/cpp/memory/monotonic_buffer_resource/monotonic - Дата доступа: 27.11.2021.

10. Официальная документация по языку программирования C++. `EBCO` [Электронный ресурс] / - cppreference.com - Режим доступа: https://en.cppreference.com/w/cpp/language/ebo. - Дата доступа: 28.11.2021.

11. Официальная документация по языку программирования C++. `select_on_container_copy_construction` [Электронный ресурс] / - cppreference.com - Режим доступа: https://en.cppreference.com/w/cpp/memory/allocator_traits/select_on_container_c - Дата доступа: 29.11.2021.

12. Официальный сайт рабочей группы по развитию C++. `P0784R7` [Электронный ресурс] / - open-std.org - Режим доступа: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0784r7.html. - Дата доступа: 30.11.2021.

13. Конференция CppCon 2017. [Электронный ресурс] / - Allocators: The Good Parts. Pablo Halpern. - Режим доступа: https://www.youtube.com/watch?v=v3dz-AKOVL8. - Дата доступа: 02.12.2021.

14. Проект OpenNet. [Электронный ресурс] / - UNIX man. - Режим доступа:

https://www.opennet.ru/cgi-bin/opennet/man.cgi?topic=mmapcategory=2. - Дата доступа: 03.12.2021.

15. Аллокатор TCMalloc. [Электронный ресурс] / - github.com/google - Режим доступа: https://github.com/google/tcmalloc. - Дата доступа: 04.12.2021.
16. Аллокатор jemalloc. [Электронный ресурс] / - github.com/jemalloc - Режим доступа: https://github.com/jemalloc/jemalloc. - Дата доступа: 05.12.2021.
17. Аллокатор LowFat. [Электронный ресурс] / - github.com/GJDuck - Режим доступа: https://github.com/GJDuck/LowFat. - Дата доступа: 05.11.2021.
18. jemalloc был удалён из стандартной библиотеки. [Электронный ресурс] / - internals.rust-lang.org - Режим доступа: https://internals.rust-lang.org/t/jemalloc-was-just-removed-from-the-standard-library/8759. - Дата доступа: 06.11.2021.
19. Официальная документация языка программирования Rust. `std::alloc` [Электронный ресурс] / - doc.rust-lang.org - Режим доступа: https://doc.rust-lang.org/std/alloc/index.html. - Дата доступа: 06.11.2021.
20. Предложение в стандарт C++ о деаллокациях с размером. [Электронный ресурс] / - open-std.org - Режим доступа: http://open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3536.html. - Дата доступа: 14.02.2022.
21. Аллокаторы внутри. [Электронный ресурс] / - habr.com - Режим доступа: https://habr.com/ru/post/645137/. - Дата доступа: 18.02.2022.
22. Официальная документация языка программирования Zig. [Электронный ресурс] / - ziglang.org - Режим доступа: https://ziglang.org/documentation/0.9.0/Memory. - Дата доступа: 19.02.2022.
23. Документация компании IBM по языку программирования COBOL. [Электронный ресурс] / - ibm.com - Режим доступа:

https://www.ibm.com/docs/en/developer-for-zos/14.2.0?topic=operations-memory-allocation. - Дата доступа: 19.02.2022.

24. Документация компании IBM по языку программирования Fortran. [Электронный ресурс] / - ibm.com - Режим доступа: https://www.ibm.com/docs/en/xffbg/121.141?topic=attributes-allocate. - Дата доступа: 19.02.2022.

25. Официальная документация языка программирования D. [Электронный ресурс] / - dlang.org - Режим доступа: https://dlang.org/library/core/stdc/stdlib/malloc.html. - Дата доступа: 19.02.2022.

26. Официальная документация языка программирования C++. [Электронный ресурс] / - en.cppreference.com - Режим доступа: https://en.cppreference.com/w/cpp/memory/construct_at. - Дата доступа: 20.02.2022.

27. Официальная документация языка программирования C++. [Электронный ресурс] / - en.cppreference.com - Режим доступа: https://en.cppreference.com/w/cpp/memory/destroy_at. - Дата доступа: 20.02.2022.

28. Статья о Mark-and-Sweep на GeeksForGeeks. [Электронный ресурс] / - geeksforgeeks.org - Режим доступа: https://www.geeksforgeeks.org/mark-and-sweep-garbage-collection-algorithm/. - Дата доступа: 21.02.2022.

29. Порождающий паттерн проектирования одиночка. [Электронный ресурс] / - refactoring.guru - Режим доступа: https://refactoring.guru/ru/design-patterns/singleton. - Дата доступа: 22.02.2022.

30. Предложение в стандарт C++ удалить минимальную поддержку сборщика мусора. [Электронный ресурс] / - open-std.org - Режим доступа: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2186r2.html. - Дата доступа: 22.02.2022.

31. Сборщик мусора в Golang. [Электронный ресурс] / - habr.com - Режим доступа: https://habr.com/ru/post/265833/. - Дата доступа: 22.02.2022.

32. Доклад Андрея Роенко о сборщиках мусора на Frontend Conf Moscow 2018. [Электронный ресурс] / - youtube.com - Режим доступа: https://www.youtube.com/watch?v=tDbRVZqwxn4. - Дата доступа: 23.02.2022.

33. Статья на сайте Корнелльского университета о копирующем сборщике мусора. [Электронный ресурс] / - cs.cornell.edu - Режим доступа: http://www.cs.cornell.edu/courses/cs312/2003fa/lectures/sec24.htm. - Дата доступа: 23.02.2022.

34. Доклад Владимира Иванова о сборщике мусора G1. [Электронный ресурс] / - youtube.com - Режим доступа: https://www.youtube.com/watch?v=iGRfyhE02lA. - Дата доступа: 28.02.2022.

35. Доклад Алексея Шипилёва о сборщике мусора Shenandoah (часть 1). [Электронный ресурс] / - youtube.com - Режим доступа: https://www.youtube.com/watch?v=c1jVn5Sm8Uw. - Дата доступа: 03.03.2022.

36. Доклад Алексея Шипилёва о сборщике мусора Shenandoah (часть 2). [Электронный ресурс] / - youtube.com - Режим доступа: https://www.youtube.com/watch?v=jaiRW1v2fjk. - Дата доступа: 03.03.2022.

37. Официальная документация к сборщику мусора EpsilonGC. [Электронный ресурс] / - openjdk.java.net - Режим доступа: http://openjdk.java.net/jeps/318. - Дата доступа: 12.03.2022.

38. Блог компании JetBrains. [Электронный ресурс] / - blog.jetbrains.com - Режим доступа: https://blog.jetbrains.com/kotlin/2021/05/kotlin-native-memory-management-update/#kn-gc. - Дата доступа: 13.03.2022.

39. Описание работы сборщика мусора в CPython. [Электронный ресурс] / - arctrix.com - Режим доступа: http://arctrix.com/nas/python/gc/. - Дата доступа: 16.03.2022.

40. Описание работы сборщиков мусора в PyPy. [Электронный ресурс] / - doc.pypy.org - Режим доступа: https://doc.pypy.org/en/latest/gc_info.html. - Дата доступа: 17.03.2022.

41. Документация ZGC. [Электронный ресурс] / - wiki.openjdk.java.net - Режим доступа: https://wiki.openjdk.java.net/display/zgc/Main. - Дата доступа: 18.03.2022.

42. C# garbage collector. [Электронный ресурс] / - medium.com - Режим доступа: https://medium.com/c-programming/c-memory-management-part-3-garbage-collection-18faf118cbf1. - Дата доступа: 20.03.2022.

43. Официальная документация Nim. [Электронный ресурс] / - nim-lang.org - Режим доступа: https://nim-lang.org/1.4.0/gc.html. - Дата доступа: 22.03.2022.

44. Официальная документация D. [Электронный ресурс] / - dlang.org - Режим доступа: https://dlang.org/spec/garbage.html#how_gc_works. - Дата доступа: 24.03.2022.

45. Модель памяти в различных языках программирования. [Электронный ресурс] / - memorymanagement.org - Режим доступа: https://www.memorymanagement.org/mmref/lang.html. - Дата доступа: 26.03.2022.

46. Предложение в стандарт C++ ввести минимальную поддержку сборщика мусора. [Электронный ресурс] / - open-std.org - Режим доступа: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2670.htm. - Дата доступа: 13.04.2022.

ПРИЛОЖЕНИЕ 1

Исходный код: https://gist.github.com/dasfex/bff975bb1e66b26823bb99f1cce019ff.

```
class Tree {
public:
  struct Node : GC::ObjHeader<2> {
    Node* left;
    Node* right;
    int val;
  };

  void Add(int v) {
    add(root.ref, v);
  }

  int Find(int k) {
    return find(root.ref, k)->val;
  }

  void Link(Tree& tree, int k) {
    tree.root.ref = find(root.ref, k);
  }

  void Cut(int k) {
    cut(root.ref, k);
  }

  void Print() {
```

```
    print(root.ref);
  }

  DECL(Node) root;

private:

  void add(Node*& node, int k) {
    if (!node) {
      node = NEW(Node);
      node->left = node->right = nullptr;
      node->val = k;
      return;
    }
    if (node->val == k) {
      return;
    }
    if (node->val < k) {
      add(node->left, k);
    } else {
      add(node->right, k);
    }
  }

  Node* find(Node* node, int k) {
    if (!node || node->val == k) {
      return node;
    }
    if (node->val < k) {
      return find(node->left, k);
    } else {
      return find(node->right, k);
    }
```

```
  }

  void cut(Node*& target, int k) {
    if (!target || target->val == k) {
      target = nullptr;
      return;
    }
    if (target->val < k) {
      cut(target->left, k);
    } else {
      cut(target->right, k);
    }
  }

  void print(Node* t, int indent = 0) {
    if (!t) { return; }

    std::cout << std::string(indent, ' ');
    std::cout << t << ' ' << t->val << std::endl;

    print(t->left, indent + 1);
    print(t->right, indent + 1);
  }
};

int main() {
  GC::Init();

  Tree tree;

  for (int x : {2, 1, 3, 6, 5, 4, 8}) {
    tree.Add(x);
  }
```

```
    Tree add;
    tree.Link(add, 3);
    std::cout << "Before GC" << std::endl;
    std::cout << add.root.ref << ' ' << GC::GetOffset()
                              << std::endl << std::endl;
    tree.Print();
    std::cout << std::endl;

    GC::Collect();
    std::cout << "After GC:" << std::endl;
    std::cout << add.root.ref << ' ' << GC::GetOffset()
                              << std::endl << std::endl;
    tree.Print();

    std::cout << std::endl;
    tree.Cut(5);
    GC::Collect();
    std::cout << "After GC:" << std::endl;
    std::cout << add.root.ref << ' ' << GC::GetOffset()
                              << std::endl << std::endl;
    tree.Print();

    std::cout << std::endl;
    tree.Cut(2);
    add.Cut(3);
    GC::Collect();
    std::cout << "After GC:" << std::endl;
    std::cout << add.root.ref << ' ' << GC::GetOffset()
                              << std::endl << std::endl;
    tree.Print();
    add.Print();
    return 0;
}
```

Результаты выполнения:

```
Before GC
0x61d0000000c8 224

0x61d000000088 2
 0x61d0000000c8 3
  0x61d0000000e8 6
  0x61d000000148 8
  0x61d000000108 5
    0x61d000000128 4
 0x61d0000000a8 1

After GC:
0x61d000000aa8 224
0x61d000000a88 2
 0x61d000000aa8 3
  0x61d000000ac8 6
  0x61d000000ae8 8
  0x61d000000b08 5
    0x61d000000b28 4
 0x61d000000b48 1

After GC:
0x61d0000014a8 160
0x61d000001488 2
 0x61d0000014a8 3
  0x61d0000014c8 6
  0x61d0000014e8 8
 0x61d000001508 1

After GC:
0x0 0
```