МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет прикладной математики и информатики

Кафедра дискретной математики и алгоритмики

Ходор Иван Андреевич

Разработка аллокаторов для динамического управления памятью

Курсовой проект
студента 4 курса 3 группы

Научный руководитель
Лукьянов И.Д.
старший преподаватель кафедры ДМА

МИНСК 2021

СОДЕРЖАНИЕ

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, однако перед рассмотрением такой сложной концепции стоит разобраться с методами управления динамической памятью. 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.

Целью данной работы является рассмотрение истории стандартного аллокатора на примере языка программирования C++, выделение его достоинств и недостатков, а также изучение стандартных стратегий для проектирования аллокаторов, их малоизвестные возможностей и популярных в промышленном программировании аллокаторов. В качестве закрепления материала спроектируем устойчивый к повреждениям памяти аллокатор с использованием рассмотренных методов, в процессе чего покажем практическую ценность следования некоторым устоявшимся в индустрии методам и рекомендациям.

Исследование аллокаторов является необходимой ступенью перед изучением методов сборки мусора, потому как непонимание базовых концепций управления памятью блокирует возможность разработки более сложных.

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:

> *... 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 очевидные 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 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

АЛЛОКАТОРЫ НА ПРАКТИКЕ

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[8]

Самым простым примером нестандартного аллокатора является `null`-аллокатор. 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[8]

Простейшей композицией является `fallback`-аллокатор:

```
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[8]

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:

```
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[8]

`Freelist` 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[8]

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. 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[8]

Довольно полезным также является `stats`-аллокатор, 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[8]

Такой аллокатор выделяет память блоками равного размера и на каждый блок имеет всего 1 бит метаинформации: занят ли блок. 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[8]

Чаще всего `BitmappedBlock` 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[8]

`Segregator` 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[8]

`Bucketizer` 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[13]

В C++17 наряду с основной моделью аллокатора была введена новая: `std::pmr::polymorphic_allocator`(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.

2.2.6 Аллокаторы в Rust

Несмотря на то, что аллокаторы изначально являются концепцией C++, они имеют место и в некоторых других языках программирования. Например, в Rust. Стоит отметить некоторые интересные моменты:

- 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].
- В Rust 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.

ГЛАВА 3

РЕАЛИЗАЦИЯ РАСПРЕДЕЛЁННОГО АЛЛОКАТОРА, УСТОЙЧИВОГО К ПОТЕРЯМ ДАННЫХ

3.1 Введение

В современном мире довольно часто приходится использовать различные подходы для распределения нагрузки между частями системы. Это бывает полезно в случаях, когда продуктом пользуется большое количество клиентов и каждая минута простоя сервиса может принести значительные убытки. Популярным решением является как можно глубокое разделение или дробление всей системы на более мелкие специализированные части – микросервисы. Например, при отказе одного микросервиса вся остальная система будет работать, и сломаная часть пользователю может даже не пригодиться: он ничего не заметит, а разработчики всё исправят, что позволит не терять прибыль и репутацию.

Похожие принципы распределения нагрузки также используются и при хранении информации: все данные хранятся не в одном экземпляре, а нескольких. Чаще всего имеется основной кластер (мастер), с которым происходят все операции, и несколько копий (реплик), которые время от времени обновляют данные на основании того, что в текущий момент находится в мастере.

3.2 Постановка задачи

Спроектируем аллокатор на основе `std::allocator`, хранящий несколько экземпляров данных, что позволит избежать потери данных при внешнем воздействии на используемые участки памяти. В этой главе пройдём путь от тривиального решения до более сложного с помощью добавления некоторых

стратегий, рассмотренных ранее.

3.3 Симуляция внешнего воздействия

В качестве внешнего воздействия в учебных целях будем использовать родительский для разрабатываемого аллокатор. Все операции с памятью он делегирует дочернему аллокатору, а сам лишь является обёрткой, которая выбирает случайный занятый блок и полностью зануляет все байты в нём. Раз в некоторое время этот аллокатор будет выбирать случайный блок и зануллять все байты в нём.

В реальной жизни таким внешним воздействием может быть действие некоторой вирусной программы, стремящейся незаметно мешать работе различных программ.

Отметим, что для такого аллокатора требуются реализации лишь методов выделения/удаления памяти и механизма повреждения памяти.

Схематично этот аллокатор выглядит так:

```
template <class T, class ChildAlloc>
class DamageAlloc {
  ChildAlloc alloc_;
  std::unordered_map<T*, size_t> indices_;
  std::vector<Blk> blocks_;
  // timer, damage
public:
  // allocate, deallocate, construct, destroy
};
```

При удалении основное время будет тратиться на поиск нужного блока, потому также храним отображение из указателя в индекс блока. В силу того, что порядок блоков не важен, достаточно поменять удаляемый блок с последним и корректно изменить индексы в `indices_`.

3.4 Базовое решение

Первой проблемой, которую требуется решить, является вопрос о том, как же хранить копии данных. Первоначальный интерфейс будет выглядеть так:

```
template <class T>
class DupAlloc;
```

Необходимо определиться с количеством копий для каждого блока памяти. Очевидно, что чем больше копий будет храниться, тем больше памяти будет потреблять аллокатор, но тем более устойчивым к повреждениям памяти он станет. Так как для корректного выбора требуется знать контекст использования аллокатора, предоставим этот выбор пользователю, чтобы он сам принял некоторое компромиссное решение:

```
template <class T, size_t N>
class DupAlloc {
  std::array<std::set<T*>, N> copies_;
  std::map<T*, std::reference_wrapper<std::set<T*>>> map_;
  ...
};
```

При каждом выделении памяти выделяется не один блок памяти, а `N`, и каждый добавляется в свою группу блоков в `map_` (отображение из указателя на блок в его группу). Теперь при удалении/конструировании/деконструировании объекта есть возможность быстро находить всю группу по указателю и проводить все необходимые операции.

3.5 Использование композиций

3.5.1 Улучшение аллокатора для упрощения композиции

Как уже говорилось выше, хороший аллокатор тот, который легко используется в качестве композиции. Чтобы следовать этой рекомендации, стоит добавить в проектируемый аллокатор возможность передать аллокатор, который будет выделять память. По умолчанию стоит использовать `Mallocator`:

```
template <class T, size_t N, class Alloc = Mallocator>
class DupAlloc { Alloc alloc_; };
```

Теперь все методы выделения/удаления памяти и конструирования/деконструирования объекта можно делегировать данному аллокатору.

Т.к. присутствует некоторый базовый аллокатор, то также можно реализовать метод `owns`: `return alloc_.owns(blk);`.

3.5.2 Affix-аллокатор

Следующая подзадача – проверка корректности данных в блоках и, если нужно, их восстановление.

Выясним корректность блоков. Необходимо иметь некоторый индикатор, который подскажет, корректна ли память в текущий момент. Хорошим решением будет использование `affix`-аллокатора для добавления префикса с данными к выделяемому блоку. В них достаточно хранить какие-то ненулевые значения. В случае, когда при проверке блока префикс является нулевым, имеем, что блок был повреждён, потому ищем в его группе корректный блок и восстанавливаем информацию. При добавлении `affix`-аллокатора код будет выглядеть следующим образом:

```
template <class T, size_t N, class Alloc = Mallocator>
class DupAlloc {
 private:
  struct CheckedRegion { // for prefix
    uint64_t eight_bytes;
  };
  using AffixAllocT = AffixAlloc<Alloc, CheckedRegion>;
  AffixAllocT alloc_;
};
```

Выясним опции проверки. Очевидно, что перебирать все блоки очень неэффективно, потому приемлемым решением будет выбор нескольких случайных блоков и проверка всех блоков из группы выбранного на корректность. Осуществлять такую проверку можно так же как и в `DamageAlloc`: по таймеру.

Так как количество блоков и промежуток проверки напрямую влияют на производительность, выбор значений для этих операций хорошо оставить пользователю, предложив некоторые значения по умолчанию. Таким образом добавления будут выглядеть следующим образом:

```
template <class T, size_t N, class Alloc = Mallocator>
class DupAlloc {
 private:
  size_t timeout_ms_ = 100;
  size_t blocks_check_ = 3;
  // timer, function for checking
 public:
  DupAlloc(size_t ms, size_t blocks)
    : timeout_ms_(ms), blocks_check_(blocks) {}
};
```

3.5.3 Freelist

Одной из самых полезных оптимизаций, как можно было заметить, является `freelist`. Так как определение некоторых параметров при его создании является задачей пользователя, то необходимо предоставить ему эту возможность и здесь:

```
template <
    class T, size_t N,
    class Alloc = Mallocator,
    size_t MinFree = sizeof(T) / 2 + 1,
    size_t MaxFree = sizeof(T),
    size_t MaxBlocks = 1024
> class DupAlloc {
 private:
  using BaseAlloc = Freelist<AffixAllocT,
                      MinFree, MaxFree, MaxBlocks>;
  BaseAlloc alloc_;
};
```

3.6 Анализ спроектированного решения

Спроектированный аллокатор решает поставленную задачу: хранение копий информации для восстановления информации при её повреждении.

Достоинства предложенного решения:

- Простое построение композиций с другими аллокаторами.

  Возможность как предоставить некоторый базовый аллокатор вместо стандартных вызовов, так и использовать спроектированный аллокатор как базовый для других аллокаторов.

- Использование `affix`-аллокатора для индикации корректности блока позволяет использовать возможности кеша для быстрого доступа к данным.

- Использование `freelist` позволяет реже обращаться к системе для получения блоков памяти.

- Различное количество гибких опций, которые пользователь может настроить по своему усмотрению для получения компромисса между безопасностью и быстродействием.

Минусы предложенного решения:

- Нетривиальная реализация, которая требует внимания к мелочам и различным крайним случаям.

- Чем больше копий хранимых блоков, тем больше памяти требует аллокатор для поддержания внутренних структур данных.

Несмотря на неоднозначность предлагаемого решения, можно сделать вывод: проектирование аллокаторов и структур, с помощью которых легко составлять композицию стратегий, значительно упрощает их использование, что позволяет не тратить время на обеспечение совместимости различных методов и быстро начинать использовать дополнительный функционал.

ЗАКЛЮЧЕНИЕ

Понимание процесса возникновения стандартного аллокатора, знание его истории помогает сменить точку зрения с ”если он существует, то является хорошим решением”на ”а что мы можем сделать лучше”? Рассмотрев различные техники построения аллокаторов, с помощью которых возможно построение абсолютно любой стратегии управления динамической памятью(что и было показано во время проектирования устойчивого к потерям данных аллокатора), и некоторые примеры из промышленного программирования, можно с уверенностью сказать, что хороший аллокатор – это про использование композиции. Потому одной из важнеших задач проектирования нового аллокатора является не только корректность и хорошие показатели потребления памяти и времени, но и простота его использования с другими стратегиями. Такое положение дел наталкивает на мысль, что в этой области ещё не всё исследовано, что означает существование направлений развития существующих средств для достижения большей эффективности.

Исследование, проведённое в данной работе, является основой для разработки сборщика мусора на языке програмиирования C++, что является довольно нетривиальной задачей в силу специфики этого языка. Это и будет целью следующей работы.

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. Официльная 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.