Оглавление

[Введение 3](#_Toc167709401)

[Актуальность 3](#_Toc167709402)

[Глава 1: Теоретическая часть 4](#_Toc167709403)

[1.1 Основные понятия 4](#_Toc167709404)

[1.2 Принцип работы пулового аллокатора 4](#_Toc167709405)

[1.3 Иные аллокаторы: 6](#_Toc167709406)

[1.4 Сравнение аллокаторов 8](#_Toc167709407)

[1.5 Вывод 9](#_Toc167709408)

[Глава 2: Ход работы 10](#_Toc167709409)

[2.1 Классы 10](#_Toc167709410)

[Заключение 15](#_Toc167709411)

[Список литературы 16](#_Toc167709412)

Введение

При разработке программного обеспечения часто возникает необходимость эффективно управлять памятью. Один из подходов к этой проблеме - использование пулового аллокатора. Пуловый аллокатор - это система выделения памяти, которая использует фиксированный размер блоков памяти (пулы) для удовлетворения запросов на выделение памяти. Этот подход позволяет уменьшить фрагментацию памяти и увеличить производительность, так как выделение и освобождение памяти происходит быстрее, чем при использовании обычных аллокаторов.

Актуальность

Актуальность данной работы обусловлена тем, что во многих приложениях, особенно в играх и мультимедийных приложениях, требуется высокая производительность и низкая фрагментация памяти. Пуловый аллокатор является эффективным средством для достижения этих целей. Кроме того, пуловый аллокатор может быть полезен при разработке встраиваемых систем, где объем доступной памяти ограничен.

Глава 1. Теоретическая часть

## 1.1 Основные понятия

Аллокатор - это система выделения памяти, которая отвечает за выделение и освобождение памяти для программы.

Фрагментация памяти - это разбиение памяти на небольшие фрагменты, которые не могут быть эффективно использованы программой.

Пул - это блок памяти, разделенный на более мелкие участки фиксированного размера, которые могут быть использованы для выделения и освобождения памяти. Пулы используются для эффективного управления памятью, позволяя быстро выделять и освобождать блоки памяти фиксированного размера, минимизируя фрагментацию памяти. Статические пулы выделяются во время компиляции и имеют фиксированный размер, в то время как динамические пулы выделяются во время выполнения программы и могут изменяться в размере в зависимости от потребностей программы.

## 1.2 Принцип работы пулового аллокатора

При запросе на выделение памяти аллокатор проверяет, есть ли свободные блоки памяти в существующих пулах. Если таковые имеются, аллокатор выделяет память из одного из них. Если свободных блоков памяти нет, менеджер пулов создает новый пул и аллокатор выделяет память из него.

При освобождении памяти аллокатор возвращает блок памяти в пул, из которого он был выделен.

**Преимущества и недостатки пулового аллокатора**

**Преимущества:**

* Уменьшение фрагментации памяти.
* Увеличение производительности за счет быстрого выделения и освобождения памяти.
* Возможность использования пользовательских аллокаторов для разных типов объектов.

**Недостатки:**

* Необходимость вручную управлять пулами и размером блоков памяти.
* Невозможность эффективно использовать память, если размер объектов сильно отличается.

## 1.3 Иные аллокаторы:

**Linear Allocator**

Linear Allocator, он же «линейный» - это самый простой вид аллокаторов. Идея состоит в том, чтобы сохранить указатель на начало блока памяти выделенному аллокатору, а также использовать другой указатель или числовое представление, которое необходимо будет перемещать каждый раз, когда выделение из аллокатора завершено. В этом аллокаторе внутренняя фрагментация сведена к минимуму, потому что все элементы вставляются последовательно (пространственная локальность), и единственная фрагментация между ними — выравнивание. Однако, Linear Allocator не поддерживает выборочное освобождение определенных блоков памяти.

**Преимущества:**

• Простота реализации.

• Хорошая пространственная локальность.

• Минимальная внутренняя фрагментация.

**Недостатки:**

• Отсутствие поддержки выборочного освобождения памяти.

• Увеличение внешней фрагментации при длительном использовании.

**Stack Allocator**

Stack Allocator - это умная эволюция линейного распределителя, которая позволяет управлять памятью, как стеком. Все так же, как и раньше, мы сохраняем указатель вместе с «header» блоком на текущий адрес памяти и перемещаем его вперед для каждого выделения. В отличие от линейного аллокатора, мы также можем переместить его назад, то есть выполнить операцию deallocate, которая линейным аллокатором не поддерживается.

**Преимущества:**

• Хорошая пространственная локальность.

• Поддержка операции deallocate.

• Быстрое выделение и освобождение памяти.

**Недостатки:**

• Увеличение фрагментации памяти при частом выделении и освобождении блоков разного размера.

• Ограничение на максимальный размер выделяемой памяти.

**«Примитивный стандартный аллокатор»**

Этот аллокатор использует алгоритм распределения памяти, схожий с алгоритмом, который используется стандартным аллокатором. В основе алгоритма лежит взаимодействие с «chunks» (участками), внутри себя он содержит указатели на начало и конец памяти, указатель на максимальный блок памяти и множество, в котором будут храниться заголовки свободных блоков.

**Преимущества:**

• Гибкость и возможность использования в различных ситуациях.

• Поддержка выделения блоков произвольного размера.

**Недостатки:**

• Худшая производительность по сравнению с другими аллокаторами.

• Увеличение фрагментации памяти при длительном использовании.

## 1.4 Сравнение аллокаторов

Каждый из аллокаторов имеет свои преимущества и недостатки. Linear Allocator прост в реализации и обеспечивает хорошую пространственную локальность, но не поддерживает выборочное освобождение памяти. Pool Allocator обеспечивает быстрое выделение и освобождение памяти, но требует больше памяти для хранения информации о свободных блоках. Stack Allocator также обеспечивает хорошую пространственную локальность и поддерживает операцию deallocate, но может иметь большую фрагментацию памяти. «Примитивный стандартный аллокатор» является более гибким и может использоваться в различных ситуациях, но может иметь худшую производительность по сравнению с другими аллокаторами.

Таблица 1 – Сравнение Аллокаторов

|  |  |  |
| --- | --- | --- |
| **Аллокатор** | **Преимущества** | **Недостатки** |
| Linear Allocator | Простота реализации, хорошая пространственная локальность, минимальная внутренняя фрагментация | Отсутствие поддержки выборочного освобождения памяти, увеличение внешней фрагментации при длительном использовании |
| Pool Allocator | Быстрое выделение и освобождение памяти  уменьшение фрагментации памяти  возможность использования пользовательских аллокаторов для разных типов объектов | Необходимость вручную управлять пулами и размером блоков памяти  невозможность эффективно использовать память, если размер объектов сильно отличается |

Продолжение таблицы 1

|  |  |  |
| --- | --- | --- |
| **Аллокатор** | **Преимущества** | **Недостатки** |
| Stack Allocator | Хорошая пространственная локальность, поддержка операции deallocate, быстрое выделение и освобождение памяти | Увеличение фрагментации памяти при частом выделении и освобождении блоков разного размера, ограничение на максимальный размер выделяемой памяти |
| «Примитивный стандартный аллокатор» | Гибкость и возможность использования в различных ситуациях, поддержка выделения блоков произвольного размера | Худшая производительность по сравнению с другими аллокаторами, увеличение фрагментации памяти при длительном использовании |

## 1.5 Вывод

Выбор аллокатора зависит от конкретных требований к программе. Если требуется высокая производительность и низкая фрагментация памяти, то лучше использовать Pool Allocator или Stack Allocator. Если требуется простота реализации и хорошая пространственная локальность, то лучше использовать Linear Allocator. Если требуется гибкость и возможность использования в различных ситуациях, то лучше использовать «Примитивный стандартный аллокатор». Однако в данной работе будет рассматриваться именно Pool Allocator.

Глава 2. Практическая реализация

Реализация пулового аллокатора будет проводиться на языке программирования C++. Для этого необходимо создать классы Pool и PoolAllocator, которые будут отвечать за управление памятью.

## 2.1 Классы

В начале необходимо создать класс Pool, который будет представлять собой блок памяти, разделенный на более мелкие участки фиксированного размера. В классе Pool будут храниться следующие данные:

* size\_t block\_size\_ - размер блока памяти в байтах.
* size\_t num\_blocks\_ - количество блоков памяти в пуле.
* char\* data\_ - указатель на начало блока памяти, выделенного для пула.
* std::vector<size\_t> free\_blocks\_ - вектор, хранящий индексы свободных блоков памяти в пуле.

Класс Pool будет иметь следующие методы (см. код 1)

Код 1 Конструктор класса Pool

|  |
| --- |
| Pool(size\_t block\_size, size\_t num\_blocks)  : block\_size\_(block\_size), num\_blocks\_(num\_blocks)  {  data\_ = new **char**[block\_size\_ \* num\_blocks\_];  **for** (size\_t i = 0; i < num\_blocks\_; i++)  {  free\_blocks\_.push\_back(i);  }  } |

конструктор, выделяющий память для пула и инициализирующий вектор свободных блоков памяти.

Код 2 Функция деструктор класса Pool

|  |
| --- |
| ~Pool()  {  delete[] data\_;  } |

деструктор, освобождающий память, выделенную для пула.

Код 3 Конструктор класса Pool

|  |
| --- |
| **void** \*Allocate()  {  **if** (free\_blocks\_.empty())  {  std::cerr << "Pool is out of memory" << std::endl;  **return** nullptr;  }  size\_t block\_index = free\_blocks\_.back();  free\_blocks\_.pop\_back();  **return** data\_ + block\_index \* block\_size\_;  } |

метод, выделяющий свободный блок памяти из пула и возвращающий указатель на него.

Код 4 Функция освобождения памяти класса Pool

|  |
| --- |
| **void** Deallocate(**void** \*ptr)  {  size\_t block\_index = (((**char** \*)ptr - data\_) / block\_size\_);  free\_blocks\_.push\_back(block\_index);  } |

метод, освобождающий блок памяти, выделенный ранее методом Allocate(), и добавляющий его индекс в вектор свободных блоков памяти.

Код 5 Функция для получения кол-ва свободных блоков памяти класса Pool

|  |
| --- |
| size\_t GetNumFreeBlocks() **const**  {  **return** free\_blocks\_.size();  } |

метод, возвращающий кол-во блоков памяти, выделенных ранее методом Allocate().

Затем необходимо создать класс PoolAllocator, который будет отвечать за управление пулами и выделение памяти из них. В классе PoolAllocator будут храниться следующие данные:

* size\_t block\_size\_ - размер блока памяти в байтах.
* size\_t num\_blocks\_per\_pool\_ - количество блоков памяти в одном пуле.
* std::vector<Pool\*> pools\_ - вектор, хранящий указатели на пулы.
* Pool\* current\_pool\_ - указатель на текущий пул, из которого выделяется память.

Класс PoolAllocator будет иметь следующие методы:

Код 6 Функция конструктор класса PoolAllocator

|  |
| --- |
| PoolAllocator(size\_t block\_size, size\_t num\_blocks\_per\_pool)  : block\_size\_(block\_size), num\_blocks\_per\_pool\_(num\_blocks\_per\_pool) {} |

конструктор, инициализирующий размер блока памяти и количество блоков памяти в одном пуле.

Код 7 Функция деструктор класса PoolAllocator

|  |
| --- |
| ~PoolAllocator()  {  **for** (**auto** pool : pools\_)  {  delete pool;  }  } |

деструктор, освобождающий память, выделенную для пулов.

Код 8 Функция выделения пула памяти класса PoolAllocator

|  |
| --- |
| **void** \*Allocate()  {  **if** (current\_pool\_ && current\_pool\_->GetNumFreeBlocks() > 0)  {  **return** current\_pool\_->Allocate();  }  current\_pool\_ = CreateNewPool();  **return** current\_pool\_->Allocate();  } |

метод, выделяющий память из текущего пула или создающий новый пул, если в текущем пуле нет свободных блоков памяти.

Код 9 Функция освобождения пула памяти класса PoolAllocator

|  |
| --- |
| **void** Deallocate(**void** \*ptr)  {  **if** (!current\_pool\_)  {  std::cerr << "Cannot deallocate memory: no current pool" << std::endl;  **return**;  }  current\_pool\_->Deallocate(ptr);  **if** (current\_pool\_->GetNumFreeBlocks() == num\_blocks\_per\_pool\_)  {  current\_pool\_ = nullptr;  }  } |

метод, освобождающий блок памяти, выделенный ранее методом Allocate(), и проверяющий, все ли блоки памяти в текущем пуле освобождены. Если все блоки памяти освобождены, то текущий пул сбрасывается.

Код 10 Функция создания дополнительного пула класса PoolAllocator

|  |
| --- |
| Pool \*CreateNewPool()  {  Pool \*pool = new Pool(block\_size\_, num\_blocks\_per\_pool\_);  pools\_.push\_back(pool);  **return** pool;  } |

вспомогательный метод, создающий новый пул и добавляющий его в вектор пулов.

Заключение

В результате выполнения данной работы был изучен принцип работы пулового аллокатора и его преимущества и недостатки по сравнению с другими типами аллокаторов. Был разработан и реализован пуловый аллокатор на языке программирования C++, который включает в себя классы Pool и PoolAllocator. Класс Pool отвечает за управление блоками памяти фиксированного размера, а класс PoolAllocator отвечает за выделение и освобождение памяти из пулов.

Реализованный пуловый аллокатор позволяет уменьшить фрагментацию памяти и увеличить производительность за счет быстрого выделения и освобождения памяти. Он может быть полезен при разработке приложений, требующих высокой производительности и низкой фрагментации памяти, а также при разработке встраиваемых систем, где объем доступной памяти ограничен.

Список литературы

1. Шилдт. Г. Самоучитель по С++, 3-е издание / Г.Шилдт; пер. с англ. – СПб: ВЧВ-Петербург, 2003 – 688с.
2. Кормен Т. Алгоритмы: Построение и анализ, 3-е издание / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн; пер. с англ. – М. : ООО «И.Д.Вильямс», 2013. – 1328с: ил. – Парал. тит. англ.
3. Топп У. Структуры данных в С++ / У. Топп, У. Форд; Пер. с англ. – М.: ЗАО «Издательство БИНОМ», 1999 – 816с: ил.
4. Das V. Principles of Data Structures Using C and C++ / V. Das, NEW AGE INTERNATIONAL (P) LIMITED, PUBLISHERS, 2006 – 376с.
5. Weiss M. Data structures and algoritm analysis in C++ / M. Weiss,Pearson Education, Inc., 2014 – 655с.
6. **Приложения**

Полный код программы:

|  |
| --- |
| #include <iostream>  #include <vector>  #include <cstdlib>  **class** Pool  {  **private**:  size\_t block\_size\_;  size\_t num\_blocks\_;  **char** \*data\_;  std::vector<size\_t> free\_blocks\_;  **public**:  Pool(size\_t block\_size, size\_t num\_blocks) : block\_size\_(block\_size), num\_blocks\_(num\_blocks)  {  data\_ = **new** **char**[block\_size\_ \* num\_blocks\_];  **for** (size\_t i = 0; i < num\_blocks\_; i++)  {  free\_blocks\_.push\_back(i);  }  }  ~Pool()  {  **delete**[] data\_;  }  **void** \*Allocate()  {  **if** (free\_blocks\_.empty())  {  std::cerr << "Pool is out of memory" << std::endl;  **return** **nullptr**;  }  size\_t block\_index = free\_blocks\_.back();  free\_blocks\_.pop\_back();  **return** data\_ + block\_index \* block\_size\_;  }  **void** Deallocate(**void** \*ptr)  {  size\_t block\_index = (((**char** \*)ptr - data\_) / block\_size\_);  free\_blocks\_.push\_back(block\_index);  }  size\_t GetNumFreeBlocks() **const**  {  **return** free\_blocks\_.size();  }  };  **class** PoolAllocator  {  **private**:  Pool \*CreateNewPool()  {  Pool \*pool = **new** Pool(block\_size\_, num\_blocks\_per\_pool\_);  pools\_.push\_back(pool);  **return** pool;  }  size\_t block\_size\_;  size\_t num\_blocks\_per\_pool\_;  std::vector<Pool \*> pools\_;  Pool \*current\_pool\_ = **nullptr**;  **public**:  PoolAllocator(size\_t block\_size, size\_t num\_blocks\_per\_pool)  : block\_size\_(block\_size), num\_blocks\_per\_pool\_(num\_blocks\_per\_pool) {}  ~PoolAllocator()  {  **for** (**auto** pool : pools\_)  {  **delete** pool;  }  }  **void** \*Allocate()  {  **if** (current\_pool\_ && current\_pool\_->GetNumFreeBlocks() > 0)  {  **return** current\_pool\_->Allocate();  }  current\_pool\_ = CreateNewPool();  **return** current\_pool\_->Allocate();  }  **void** Deallocate(**void** \*ptr)  {  **if** (!current\_pool\_)  {  std::cerr << "Cannot deallocate memory: no current pool" << std::endl;  **return**;  }  current\_pool\_->Deallocate(ptr);  **if** (current\_pool\_->GetNumFreeBlocks() == num\_blocks\_per\_pool\_)  {  current\_pool\_ = **nullptr**;  }  }  };  **int** main()  {  PoolAllocator allocator(16, 100);  **int** \*a = (**int** \*)allocator.Allocate();  std::cout << "a is alocated\n";  **int** \*b = (**int** \*)allocator.Allocate();  std::cout << "b is alocated\n";  allocator.Deallocate(a);  std::cout << "a is dealocated\n";  allocator.Deallocate(b);  std::cout << "b is dealocated\n";  std::cout << "end of program\n";  **return** 0;  } |

Исправления

* Добавить вывод программы (показать как аллакатор хранит память)
* Код подписывать снизу
* Добавить см. код
* В программе необходимо наследовать класс аллокатора от шаблонного, с виртуальными функциями