Лекция 6. Управление памятью

ИУ8

October 6, 2016

Содержание

На текущей лекции рассмотрим:

- some news about new, delete
- allocators
- memory manager models
- boost::pool
- SmallObjectAllocator by Alexandrescu

Для затравки

Какой вариант работает быстрее?

```
1 int * arr[100];
2 int * p = malloc(100 * 4);
3 for(int i = 0; i < 100; i++)
4   arr[i] = p + i;
```

```
1 int * arr[100];
2
3 for(int i = 0; i < 100; i++)
4   arr[i] = malloc(4);
```

Конечно же левый.

Как работает new?

- выделяет память под объект
- вызывает конструктор объекта
- возвращает указатель на выделенную память

Как работает delete?

- получает указатель на память, которую необходимо очистить
- вызывает деструктор объекта
- освобождает память

Example

```
1 struct T {
2   T() { std::cout << "T::ctor";}
3 };
4
5 T * ptr = new T();
6 delete ptr;
```

Переопределение new и delete

C++ позволяет переопределять методы **new** и **delete** для классов.

Example

```
1  struct T {
2    T() { std::cout << "T::ctor" << std::endl; }
3    static void* operator new(std::size_t size) {
4      auto p = ::operator new(size);
5      std::cout << "TFoo::new(" << size << ") " << p << std::endl;
6      return p;
7    }
8    static void operator delete(void* p) {
9      std::cout << "TFoo::delete(" << p << ")" << std::endl;
10     if (!p) return;
11     ::operator delete(p);
12   }
13 };
14
15 T * ptr = new T();
16 delete ptr;
```

Что можно еще делать с new?

C++ позволяет объявлять новые **operator new** и **operator delete**.

```
1  struct T {
2    T() { std::cout << "T::ctor";}
3  };
4
5  void *operator new (size_t cnt, const std::string &s) {
6    std::cout << s << std::endl;
7    return ::operator new(cnt);
8  }
9
10 TFoo * ptr = new (std::string("some bedug message")) TFoo;
11 delete ptr;
```

Оператор new, не бросающий исключение

Если требуется, чтобы оператор new не бросал исключение в случае нехватки памяти, а возвращал ноль, можно использовать оператор new с параметром std::nothrow. Этот параметр имеет тип std::nothrow_t.

```
1 void *operator new(size_t size, const std::nothrow_t &nt);
```

```
1 Type * ptr = new(std::nothrow) Type;
2 if(ptr != nullptr) {
3   // using ptr
4 }
```

Задача

В у меня есть статически выделенная память. Хочу, чтобы объект моего класса разместился в этой памяти.

Вызов принят

```
1  struct TFoo {
2    TFoo(){ std::cout << "TFoo::TFoo" << std::endl; }
3    ~TFoo(){ std::cout << "TFoo::~TFoo" << std::endl; }
4  };
5
6  constexpr int memorySize = 1000;
7  static_assert(memorySize > sizeof(TFoo), "too little memory");
8  char static_data[memorySize];
9
10 int main() {
11   char * data = static_data;
12     TFoo *foo = new (data) TFoo;
13     data += sizeof(TFoo);
14     foo->~TFoo();
15     return 0;
16 }
```

placement new

В C++ определен, так называемый, placement new, который **НЕ выделяет** память, а только создает объект, в области памяти, которая передана в качестве аргумента.

```
1 void* operator new( std::size_t count, void* ptr);
```

Таким образом, можно конструировать объекты в известной области памяти. Это память может быть выделена любым способом.

Распределитель памяти

В некоторых частях стандартной библиотеки языка C++ используются специальные объексты для выделения и освобождения памяти, которые называются **распределителями памяти**.

Распределители памяти используются как абстракция, преобразующая запросы на выделение памяти в физическую операцию её выделения.

В стандартной библиотеке C++ определен распределитель памяти по умолчанию

```
1 namespace std{
2   template<class T>
3   class allocator;
4 }
```

Он использует стандартные механизмы new и delete, но время вызова операторов остается не определенным.

Где используются распределители памяти

```
1  template <class T, class Alloc = allocator<T>>
2  class vector;
3
4
5  template <class T, class Alloc = allocator<T>>
6  class list;
7
8
9  template <class Key, class T,
10          class Hash = hash<Key>, class Pred = equal_to<Key>,
11          class Alloc = allocator<pair<const Key,T>> >
12 class unordered_map;
```

Все стандартные контейнеры используют распределители памяти для выделения динамической памяти.

С точки зрения прикладного программиста использование разных распределителей памяти не является проблемой. Для этого достаточно передать распределитель памяти как шаблонный параметр.

Основные операции распределителей памяти

- **allocate(size_t N)** выделяет память для N элементов
- **construct(void * p, Args &&...)** инициализирует элемент, на который указывает p
- **destroy(void *p)** уничтожает элемент, на который указывает p
- **deallocate(void * p, size_t N)** освобождает память p

В C++11 для создания собственного распределителя памяти требуется определить только конструктор, деструктор, allocate, deallocate. До появления C++11 распределители памяти должны были иметь намного больше функций.

По умолчанию в C++11 функции construct использует placement new, а destroy явно вызовает деструктор.

Ссылка на пример распределителя в C++11

Использование алокаторов

Если вы реализуете библиотеку, то у вас может возникнуть желание позволить пользователю вашей библиотеки использовать различные распределители памяти.

Рассмотрим пример, как можно реализовать std::vector с возможностью использования распределителя памяти

```
1  template<class T, class Alloc>
2  vector<T, Alloc>::vector(size_type num, const T& val,
3             const Alloc &a)
4    : alloc(a)
5  {
6    elemns = allocator_traits<Alloc>::allocate(alloc, num);
7    // or elems = alloc.allocate(num);
8
9    for(size_type i = 0; i < num; ++i) {
10     allocator_traits<Alloc>::construct(alloc, &elems[i], val);
11     // or alloc.construct(&elems[i], val);
12   }
13 }
```

Управление памятью

Распределители памяти используются как абстракция, но под этой абстракцией скрывается реализация определенной модели работы с памятью.

Различные программыные системы имеют свои собственные требования к используемым моделям управления памятью. Поэтому время от времени требуется реализовывать механизны управления памятью удовлетворяющие этим требованиям.

Требования к моделям управления памятью

- скорость выделения, освобождения памяти
- размер занимаемой памяти для службных нужд
- безопасность
- оптимизация под конкретное "железо"
- оптимизация под конкретные данные

Упрощенная модель управления памятью

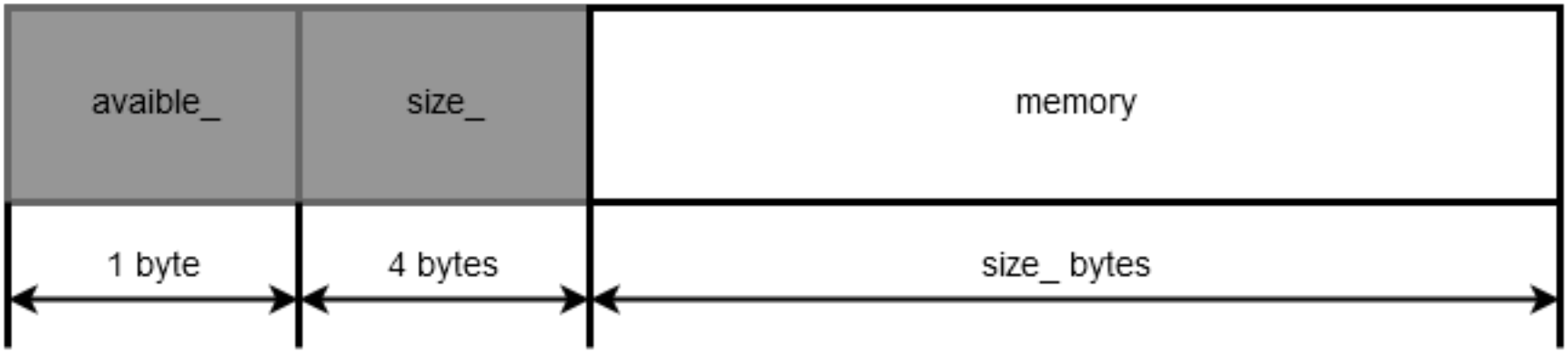
Универсальность механизма управления памятью в языке C++ может стать причиной неэффективности использования памяти.

Стандартный распределитель управлей кучей, что часто вынуждает его занимать дополнительную память (от 4 до 32 байт).

Стандартный механизм распределения памяти управляет пулом байтов и может выделять из него участнки памяти любого размера. В качестве вспомогательной структуры может применяться простой блок управления.

```
1 struct {
2   size_t size_;
3   bool avaible_;
4 };
```

Для экономии размера памяти, используемых для служебных нужд, можно для размера выделить 31 бит, а один бит оставить для обозначения доступности памяти

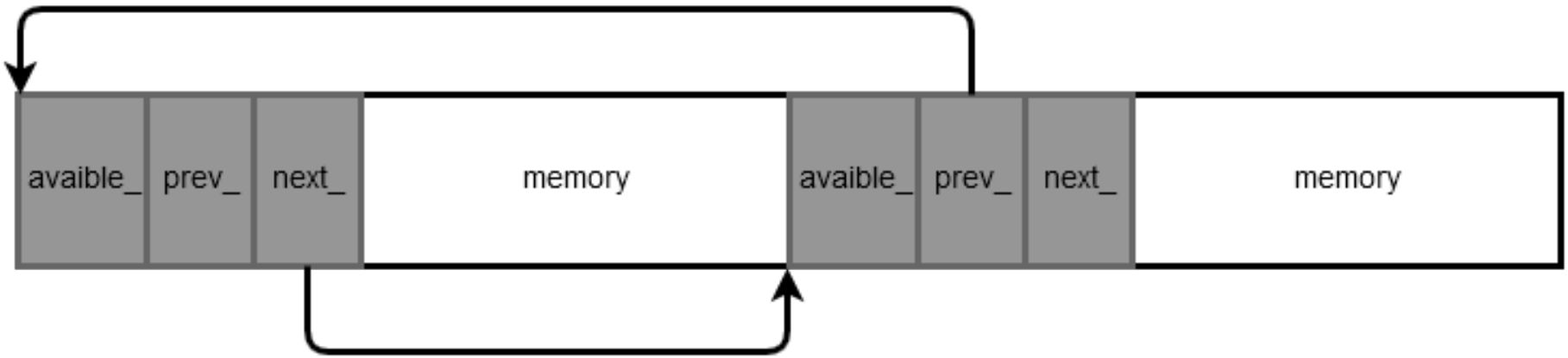


В начале работы программы в памяти располагается только одна структура. По мере выделения памяти появляются новые блоки. При новом запросе на выделение памяти, последовательно проверяются все доступные блоки необходимого размера.

Освобождение блока памяти приводит к очередному поиску предыдущего свободного блока и изменение его размера.

Можно получить постоянное время освобождение памяти, путем изменения служебной структуры.

```
1 struct MCB {
2   bool avaible_;
3   MCB * prev_;
4   MCB * next_;
5 };
```



При такой реализации там не требуется поле для хранения размера блока памяти.

boost::pool

В библиотеке boost есть несколько классов для управления памятью.

- boost::simple_segregated_storage
- boost::singleton_pool
- boost::object_pool
- boost::(fast_)pool_allocator
- boost::fast_pool_allocator

Все классы boost::pool основаны на классе boost::simple_segregated_storage.
Это очень просто класс, подразумевается, что он должен быть самым быстрый и компактный класс для управления памятью.

Using boost::simple_segregated_storage

```
1  #include <boost/pool/simple_segregated_storage.hpp>
2  #include <vector>
3  #include <cstddef>
4
5  int main()
6  {
7    boost::simple_segregated_storage<std::size_t> storage;
8    std::vector<char> v(1024);
9    storage.add_block(&v.front(), v.size(), 256);
10
11   int *i = static_cast<int*>(storage.malloc());
12   int *j = static_cast<int*>(storage.malloc_n(1, 512));
13
14   storage.free(i);
15   storage.free_n(j, 1, 512);
16 }
```

Интерфейс boost::simple_segregated_storage

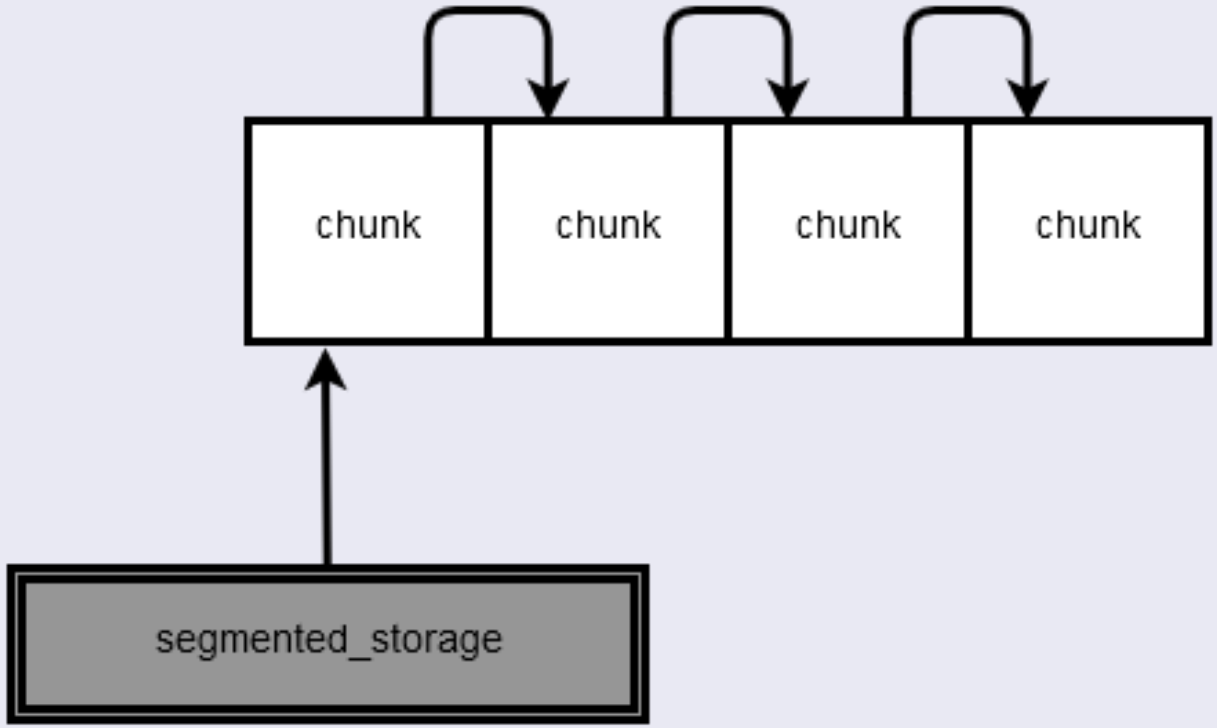
- **add_block** - метод, для добавления новой памяти в хранилище
- **malloc** - метод, который выделяет из хранилища блок памяти. возвращает указатель на выделенный блок
- **free** - метод, который возращает в хранилище блок памяти, выделенный через метод **malloc**

simple_segregated_storage низкоуровневый класс обычно не используется в программах. Более того, класс обладает рядом особенностей, которые надо учитывать и обходить стороной. Поэтому непосредственное использование данного класса не рекомендуется

```
1 boost::simple_segregated_storage<std::size_t> storage;
2 std::vector<char> v(1024);
3 storage.add_block(&v.front(), v.size(), 256);
4
5 int *j = static_cast<int*>(storage.malloc_n(1, 4096));
6 // j != nullptr.
```

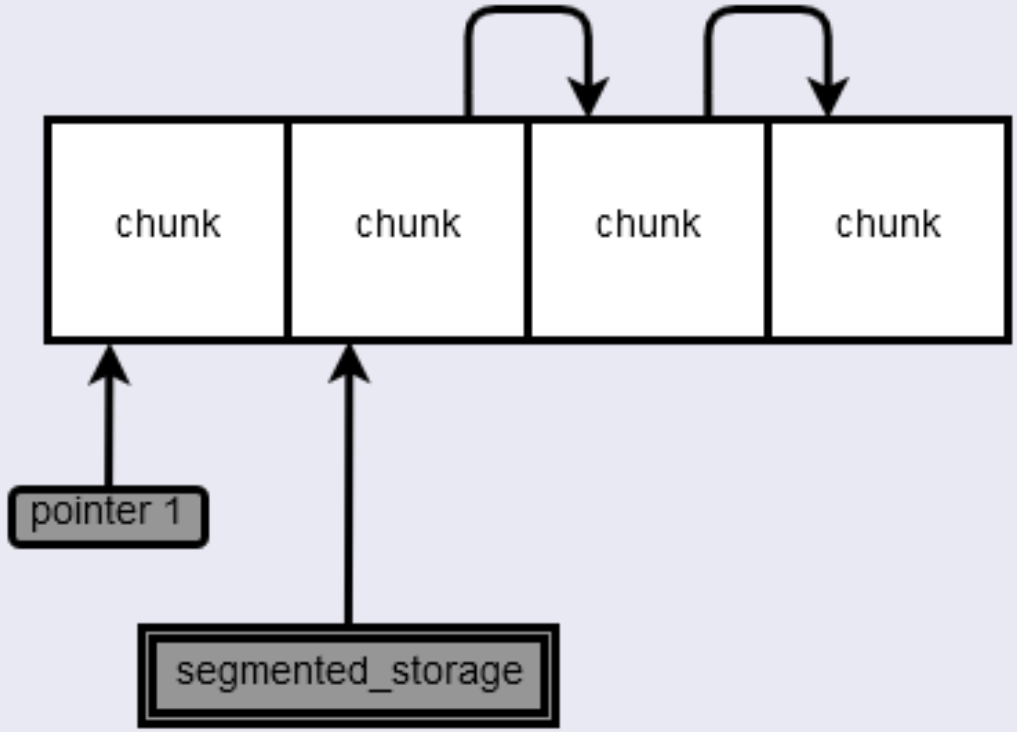
Как устроен boost::simple_segregated_storage

```
1 storage.add_block(&v.front(), v.size(), 256);
```



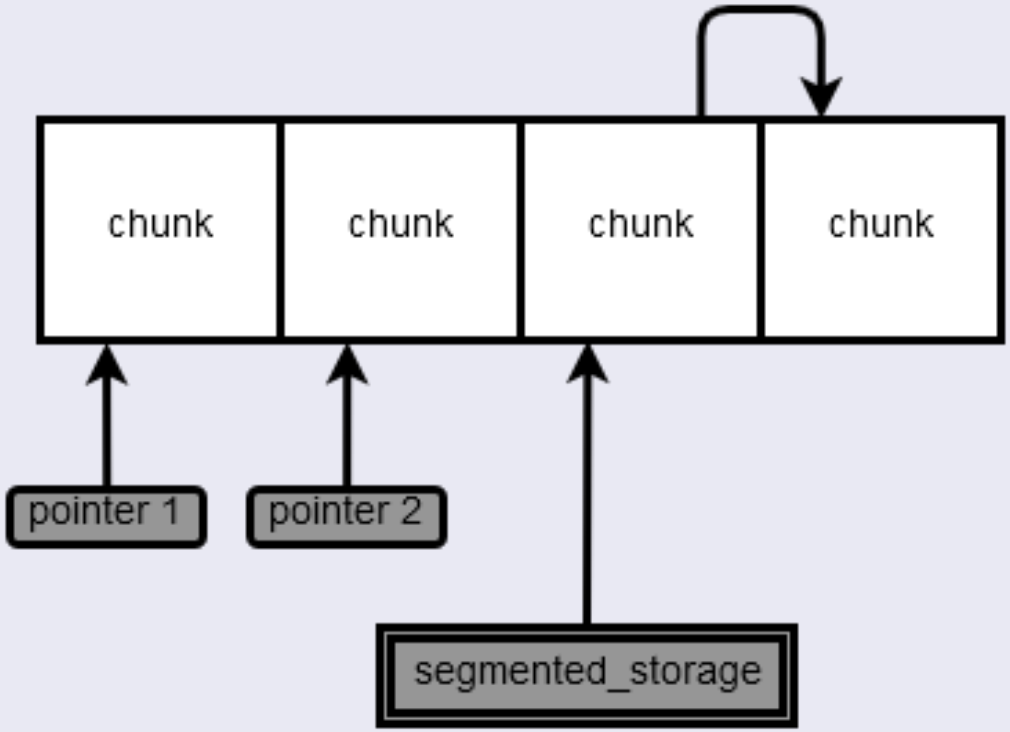
Как устроен boost::simple_segregated_storage

```
1 int * pointer1 = static_cast<int*>(storage.malloc());
```



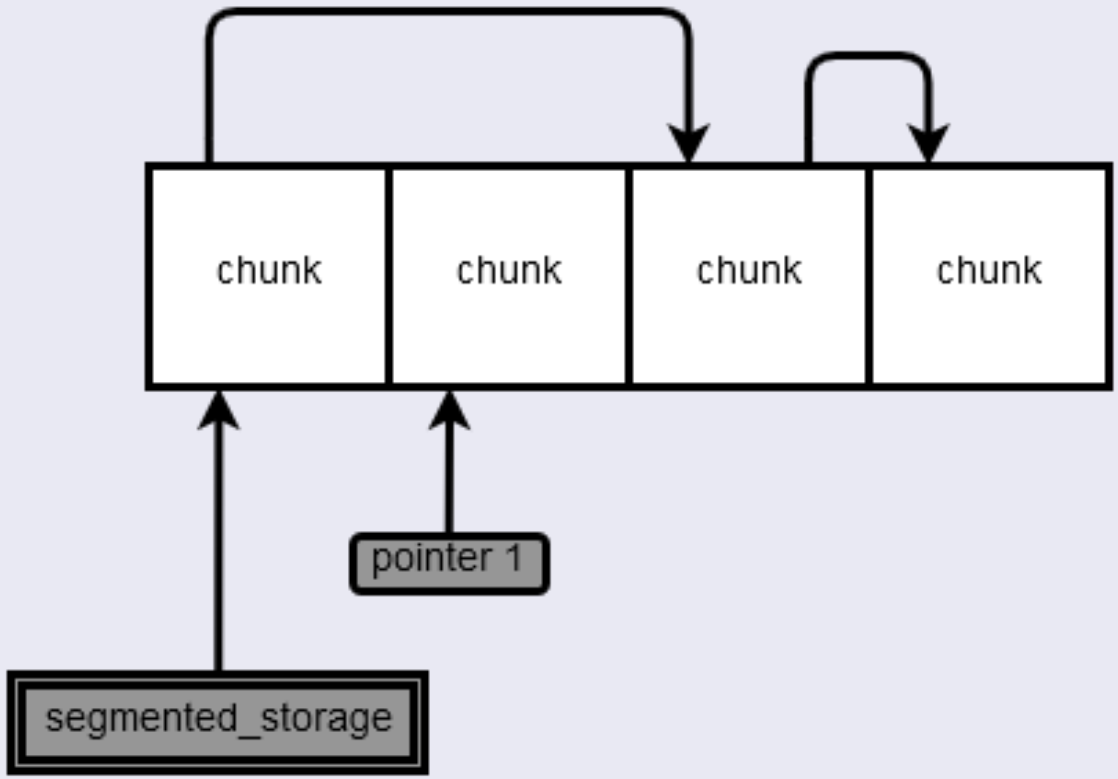
Как устроен boost::simple_segregated_storage

```
1 int * pointer2 = static_cast<int*>(storage.malloc());
```



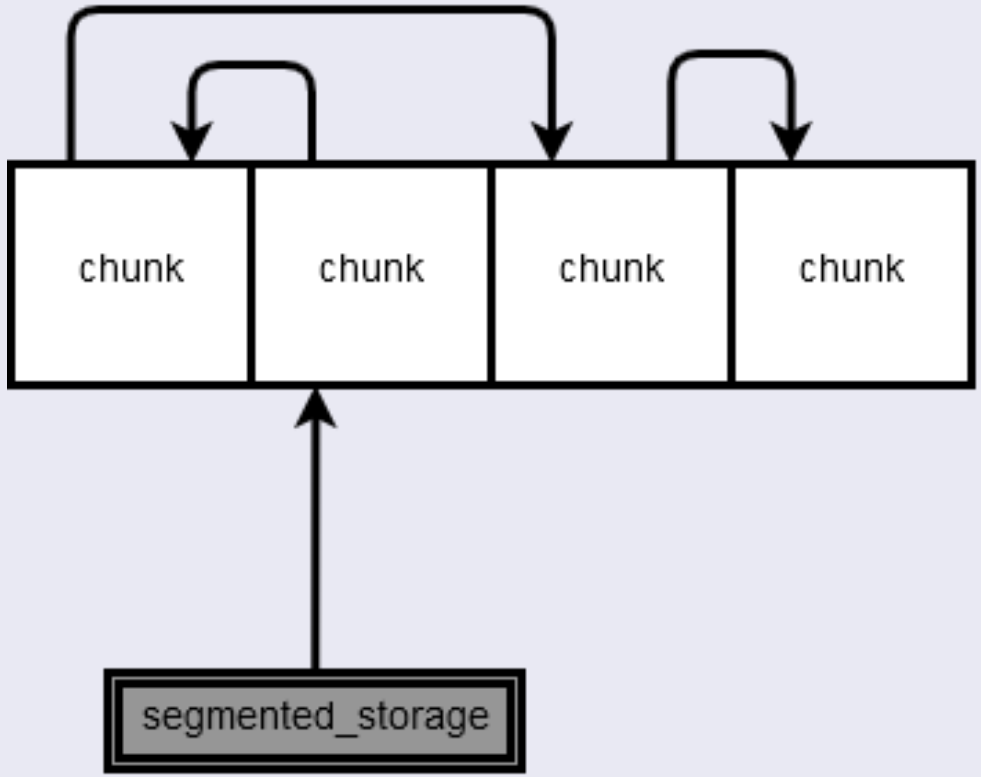
Как устроен boost::simple_segregated_storage

```
1 storage.free(pointer1);
```



Как устроен boost::simple_segregated_storage

```
1 storage.free(pointer2);
```



boost::singleton_pool

Класс **boost::singleton_pool** является более выкоуровневым механизмом для управления памятью.

```
1  struct int_pool {};
2  using singleton_int_pool = boost::singleton_pool<int_pool, sizeof(int)>;
3
4  int main() {
5    // let's allocate memory for a single element
6    int *i = static_cast<int*>(singleton_int_pool::malloc());
7
8    // let's allocate memory for 10 elements
9    int *j = static_cast<int*>(singleton_int_pool::ordered_malloc(10));
10
11   // releases all memory blocks that aren't used
12   singleton_int_pool::release_memory();
13   // releases all memory blocks
14   singleton_int_pool::purge_memory();
15 }
```

Описание типа

У класса **boost::singleton_pool** 6 шаблонных аргумента

- Tag - некий идентификатор, по которому можно получить доступ до конкретного объекта класса singleton_pool
- RequestedSize - требуемый размер для выделения памяти под один "элемент"
- UserAllocate - пользовательский аллокатор. По умолчанию default_user_allocator_new_delete.
- Mutex - тип объекта для синхронизации потоков
- NextSize - первоначальный объем пула
- MaxSize - максимальное количество выделяемых chunks при запросе на выделение памяти

Особенности singleton_pool

Возможно создать несколько экземпляров boost::singleton_pool. Благодаря первому шаблонному параметру Tag, существует возможность работать с несколькими экземплярами, даже при условии что RequestedSize одинаковый.

Класс boost::singleton_pool выделяет память автоматически, не требуется, как в случае с boost::simple_segregated_storage, предоставлять память.

Функции release_memory и purge_memory возвращают память обратно ОС. Чтобы освободить память и оставить её в boost::singleton_pool необходимао использовать free или ordered_free.

Функции malloc и ordered_malloc возвращают void * поэтому приходится явно использовать приведение к нужному типу указателя.

boost::object_pool<T>

Библиотека boost предоставляет еще класс для управления памятью - boost::object_pool

В отличие от boost::singleton_pool и boost::simple_segregated_storage объекты класса boost::object_pool знают для каких типов выделять память. Поэтому нет нужды явно использовать приведение типов.

Кроме того, благодаря этому "знанию" в boost::object_pool существует метод construct одновременного выделения памяти и вызова конструктора и метод destroy для вызова деструктора и освобождения памяти.

Еще одна особенность boost::object_pool в возможности регулирования размера блока, запрашиваемого у ОС.

```
1 void set_next_size(const size_type next_size);
```

Аллокаторы boost

Для использования механизмов управления памятью, описанных выше, boost имеет два распределителя памяти, удовлетворяющих требованиям стандартной библиотеки. Следовательно их можно использовать в качестве аллокаторов для стандартных контейнеров.

Эти распределители памяти реализованы в классах boost::pool_allocator и boost::fast_pool_allocator.

```
1 int main() {
2   std::vector<int, boost::pool_allocator<int>> v;
3   for (int i = 0; i < 1000; ++i)
4     v.push_back(i);
5
6   v.clear();
7   boost::singleton_pool<boost::pool_allocator_tag, sizeof(int)>::
8     purge_memory();
9 }
```

Обоснование оптимизации для работы с малыми объектами

Зачастую стандартные механизмы управления памятью не оптимизированы для работы с объектами малого размера.

Однако, используются небольшие объекты не так и редко.

Рассмотрим механизмы управления памятью для оптимизации работы с малыми объектами, реализованные Александреску в библиотеке Loki.

Принципы построения моделей управления памятью реализованные в boost и в Loki схожи.

Слоенный пирог классов

Распределитель памяти для небольших объектов работает с четырехслойной структурой:

- **SmallObject**
  - предоставляет функциональные возможности на уровне объектов
- **SmallObjAllocator**
  - размещает объекты разного размера
  - конфигурацию параметров можно изменять
- **FixedAllocator**
  - размещает объекты одного размера
- **Chunk**
  - размещает объекты одного размера
  - максимально возможное количество объектов фиксированно

Chunk

Каждый объект класса **Chunk** содержит участок памяти состоящий из фикцированного числа блоков, и управляет ими.

Размер и количество блоков указываются при создании Chunk

Если в объекте Chunk больше не осталось свободных блоков, то функция выделения памяти возвращает nullptr.

В некотором роде Chunk это упрощенный boost::simple_segregated_storage

Класс Chunk

```
1 struct Chunk {
2   void Init(size_t blockSize, byte blocks);
3   void * Allocate(size_t blockSize);
4   void Deallocate(void * p, size_t blockSize);
5   byte * data_;
6   byte firstAlailableBlock_;
7   byte blocksAvailable_;
8 };
```

FixedAllocator

Класс **FixedAllocator** предназначен для управления Chunk

FixedAllocator также как и объекты Chunk работает с блоками фиксированного размера, но количество этих блоков не ограничено.

Для этого FixedAllocator содержит массив Chunk-ов, в который добавляет новый Chunk, если все остальные заняты.

При запросе на выделение памяти FixedAllocator ищет в массиве Chunk, в котором есть свободные блоки.

Однако, для ускорения работы FixedAllocator хранит указатель на последний использованный для выделения блока Chunk. Если же в этом Chunk нет свободных блоков, только тогда происходит поиск во всем массиве, а указатель менятся на найденный Chunk

Трюки в Deallocate

Метод FixedAllocator::Deallocate(void * p) освобождает блок в соответствующем Chunk.

Но при вызове Deallocate неизвестно к какому именно Chunk относится освобождаемая память. Для определения, какой именно Chunk отвечает за освобождаемый блок, FixedAllocator происходит поиск по массиву Chunk-ов.

Это осуществляется проверкой лежит ли аргумент функции Deallocate в диапазоне *[c.data_, c.data_ + blockSize_ *numBlocks_]*, где c - это Chunk из массива

Для ускорения работы Deallocate класс FixedAllocator содержит указатель pDeallocChunk_ на последний Chunk, в котором происходило освобождение блока

Если освобождаемый блок не принадлежит Chunk, на который ссылает pDeallocChunk_, то сперва выполняется поиск подходящего Chunk среди соседей pDeallocChunk.

SmallObjAllocator

Класс **SmallObjAllocator** позволяет размещать в памяти объекты любого размера, объединяя в себе несколько различных объектов FixedAllocator

```
1 class SmallObjAllocator {
2   vector<FixedAllocator> pool_;
3 public:
4   SmallObjAllocator(size_t chunkSize, size_t maxObjectSize);
5   void * Allocate(size_t numBytes);
6   void Deallocate(void * p, size_t size);
7 };
```

Получив запрос на выделение памяти, объект класса SmallObjAllocator переадресует запрос наиболее подходящему объекту **FixedAllocator** либо оператору **new**.

Параметр size в методе Deallocate позволяет ускорить поиск необходимого FixedAllocator

SmallObject

Верхнем слоем в пироге классов занимает класс **SmallObject**.

```
1 struct SmallObject {
2   virtual ~SmallObject() = default;
3   static void * operator new(std::size_t size);
4   static void operator delete(void * p, std::size_t size);
5 }
```

Оператор new вызывает функцию SmallObjAllocator::Allocate(size)

Оператор delete вызывает функцию SmallObjAllocator::Deallocate(p, size)

Использовать SmallObject крайне просто

```
1 class MySmallClass : public SmallObject {
2   // implementation of our functional class
3 };
```

Так объекты MySmallClass автоматически будут использовать механизм управления памятью предназваченый для малых объектов.

Компилятор наше всё

В классе SmallObject используется возможность C++, которая позволяет переопределить оператор **delete**, который принимает два аргумента: указатель на освобождаемую память и размер освобождаемого объекта

Эта возможность формулируется как: If defined (21), and if (17) is not defined, called by the usual single-object delete-expressions if deallocating an object of type T.

```
1 void T::operator delete(void* ptr); (17)
2 void T::operator delete(void* ptr, std::size_t sz); (21)
```

Таким образом компилятор сам позаботится о передаче размера блока, который требуется для функции SmallObjAllocator::Deallocate

The end

Самостоятельное изучение

- реализовать распределитель памяти удовлетворяющи требованиям стандартной библиотеки и оптимизированный для работы с малыми объектами

Список литературы

- Memory Management Reference
- Александреску - Современное проектирование на C++
- Loki library
- The Boost C++ Libraries : Boost.Pool
- operator delete
- operator new
- Перегрузка операторов new и delete