Лекция 7. Управление памятью

ИУ8

December 5, 2017

Содержание

На текущей лекции рассмотрим:

- some news about new, delete
- stl allocators
- boost::pool
- memory manager models
- TCMalloc

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);
```

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

Есть выделенная память. Необходимо, чтобы объект класса разместился в этой памяти.

```
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 объекты для выделения и освобождения памяти, которые называются **распределителями памяти** (аллокаторами).

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 неопределенным.

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.

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 allocate, deallocate.

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

Аллокаторы boost

Библиотека 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)>::
        purge_memory();
8 }
```

Различные программные 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

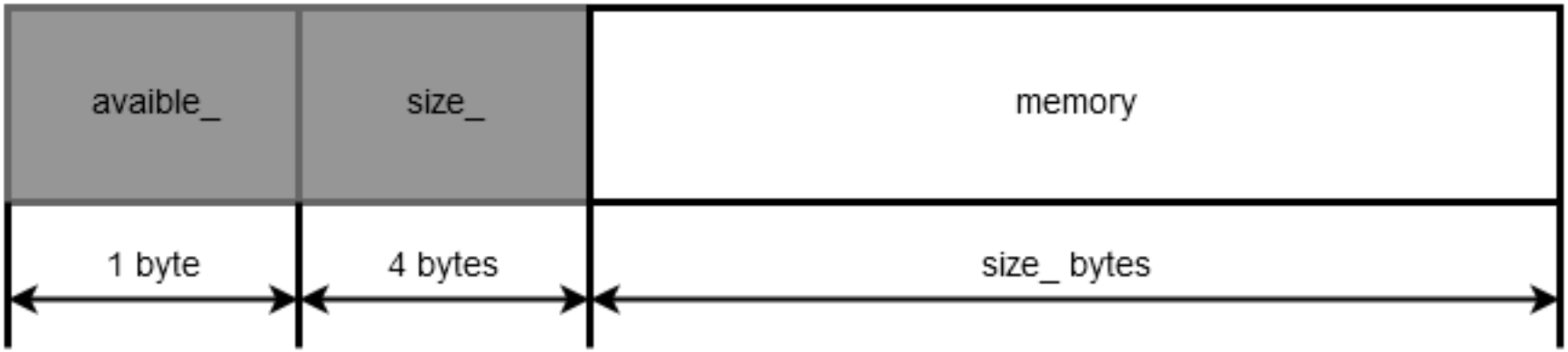
- скорость выделения/освобождения памяти
- размер занимаемой памяти для службных нужд
- оптимизация под конкретное "железо"
- оптимизация под конкретные данные

Упрощенная модель управления памятью

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

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

```
1 struct {
2   size_t size;
3   bool avaible;
4 };
```

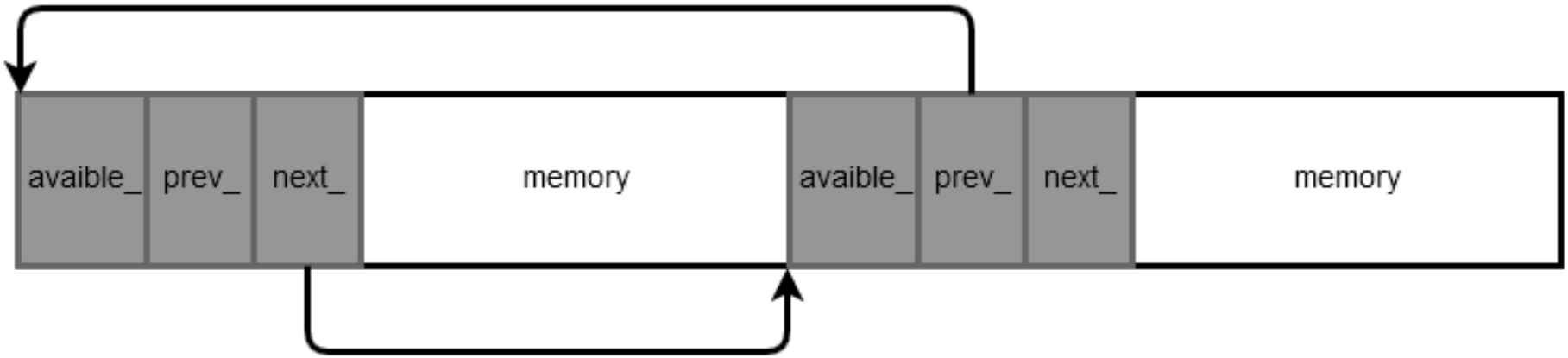


В начале работы программы в памяти располагается только одна структура. По мере выделения памяти появляются новые блоки. При новом запросе на выделение памяти последовательно проверяются все доступные блоки необходимого размера.

Освобождение блока памяти приводит к очередному поиску предыдущего свободного блока и изменению 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 };
```



Подходы к реализации эффективных аллокаторов

Основные проблемы

- фрагментация памяти
- параллелизм

Блоки разного размера

В основном для выделения памяти для "больших" объектов используют стандартные функции выделения физической памяти. Для выделения памяти под объекты малого размера используют более сложные механизмы.

Подходы к реализации эффективных аллокаторов

Fixed size allocator

Предположим, что программе всегда требуются блоки одного и того же размера. Тогда можно сделать стек свободных блоков, из которого будет выделяться память под объекты. При освобождении блоки будут добавляться в стек свободных блоков. Все это обеспечит константное время для выделения/освобождения памяти.

Так как приложениям требуется работа с памятью разных размеров, то используют несколько экземплятор fixed size allocator, которые работают со своими размерами блока.

Когда требуется выделить N байт, аллокатор ищет экземплятор fixed size allocator с размером блока, больше или равным N.

Подходы к реализации эффективных аллокаторов

Многопоточность

Самый простой подход добавить поддержку многопоточности в аллокатор - это добавить mutex. Недостатки такого решения очевидны.

Чтобы как-то нивелировать недостатки, связанные с мьютексом, создают локальный для каждого потока кэш.

TCMalloc

На этой идее основан TCMalloc от Google (TC - Thread Caching). Основная идея: для каждого потока есть свой стек свободных блоков, и только в случае переполнения этого стека блоки кладутся в глобальный стек свободных блоков. Если надо выделить блок, который все еще находится в локальном стеке, то не требуется дополнительного времени на синхронизацию.

The end

Самостоятельное изучение

- реализовать распределитель памяти удовлетворяющи требованиям стандартной библиотеки и оптимизированный для работы с объектами размера 128, 256 байт

Список литературы

- Ссылка на пример распределителя в C++11
- 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