МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«Київський політехнічний інститут»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра СКС

Лабораторна робота №1

з дисципліни

системне програмне забезпечення

на тему:

**«Аллокатор пам’яті загального призначення »**

Виконав студент Перевірив:

Групи КВ-64 \_\_\_\_\_\_\_\_\_\_\_

### Подольський Сергій Валентинович

залікова книжка № КВ6415 \_\_\_\_\_\_\_\_\_\_\_\_\_(бали)

#### VІІ семестр

Київ-2009

1. **Краткая теорія и постановка задачи**

Аллокатор памяти общего назначения должен выполнять, по крайней мере, три задачи: выделение блока памяти заданного размера, освобождение выделенного блока памяти и изменение размера выделенного блока памяти. Эти функции составляют интерфейс аллокатора памяти общего назначения:

* void \*mem\_alloc(size\_t size);

функция должна выделить блок памяти заданного размера в size байт. Если блок памяти был выделен успешно, то вернуть адрес начала этого блока, в противном случае вернуть NULL.

* void \*mem\_realloc(void \*addr, size\_t size);

функция должна изменить размер блока памяти с адресом addr до size байт. При этом содержимое (всё или часть) старого блока памяти может быть перенесено в другой блок памяти. Если удалось изменить размер блока памяти, то функция должна вернуть адрес нового блока памяти, иначе вернуть NULL и не разрушить старый блок памяти. Если addr равен NULL, то вызов функции аналогичен вызову mem\_alloc(size).

* void mem\_free(void \*addr);

функция должна освободить прежде выделенный блок памяти.

Несколько замечаний по семантике использования этих функций.

Функцию mem\_realloc() можно использовать как для уменьшения, так и для увеличения размера блока. Эта функция может выделить новый блок памяти по новому адресу, при этом содержимое старого блока всегда копируется в новый блок (всё в случае увеличения или не изменения размера блока, или часть в случае уменьшения размера). Если новый блок успешно выделен, то блок памяти по старому адресу считается недействительным. Если блок нового размера выделить не удалось, то возвращается NULL, при этом старый блок памяти должен быть доступен для использования.

Функция mem\_free() освобождает выделенный блок памяти, после вызова этой функции этот блок считается недействительным и не может использоваться приложением.

Если память выделена функциями mem\_alloc() или mem\_realloc(), то её нельзя переносить куда либо до вызова mem\_free() или mem\_realloc(), если mem\_realloc() вернула другой адрес.

Построение эффективного аллокатора общего назначение является не простой задачей. Сложность связана с тем, что не известна последовательность вызова функций выделения, освобождения и изменения размеров блоков памяти. Поэтому говорить об оптимальном, самом лучшем или идеальном решении нельзя.

Существуют несколько методов построения аллокаторов общего назначения. Все эти методы работают следующим образом. Во время инициализации аллокатора запрашивается некоторая область памяти у ОС. При запросе на выделение блока памяти, в этой области памяти ищется свободный блок памяти нужного размера, помечается как занятый и его адрес возвращается приложению. При запросе на освобождение блока памяти, этот блок помечается как свободный. Аналогично реализуется функция изменения размера блока. Если выделенной области памяти недостаточно, то выделяется ещё одна область памяти.

Рассмотрим базовый вариант одного из методов построения аллокатора общего назначения.

Каждый блок памяти состоит из заголовка фиксированного размера и данных. В заголовке указывается размер текущего блока, размер предыдущего блока и признак занятости блока. Выбирается область памяти, которая содержит хотя бы один свободный блок. Далее в найденной области памяти сканируются все блоки по порядку, и выбирается первый подходящий свободный блок. Найденный блок разделяется на два, один помечается как свободный, а второй помечается как занятый и адрес его данных возвращается приложению. Если в найденной области памяти не удалось найти свободный блок достаточного размера, то производится поиск в другой области памяти. При освобождении блока производится склейка с левым и правым свободными блоками.

Оценим этот алгоритм. Если принять, что все блоки выровнены на границу 4 байтов (здесь 1 байт это 1 октет), то размер заголовка на 32 битной системе может быть равен 8 байтам. Один бит заголовка показывается занятость блока, остальные биты показывают размер текущего и следующего блоков в байтах. Для N блоков потребуется (N\*8) байт дополнительной информации. Также потребуется память для хранения статистики и указателей на области памяти. В худшем случае при поиске свободного блока потребуется проверить N заголовков. Освобождение блока происходит быстро, для этого достаточно проверить заголовки левого и правого соседних блоков. Если выполнить 1000 пар запросов на выделение 10 байт и 20 байт, а затем освободить все блоки по 10 байт, то суммарная свободная память будет иметь размер ≥18000 байт, но для выделения нового блока в 20 байт возможно потребуется новая область памяти.

## Задание на работу

Разработать аллокатор общего назначения, используя за основу описанный выше базовый вариант алгоритма, принимая во внимания следующие условия:

1. Области памяти можно выделять любым доступным способом.
2. Функции mem\_alloc(), mem\_realloc() и mem\_free() должны соответствовать приведенным выше прототипам.
3. Адреса памяти, возвращаемые функциями mem\_alloc() и mem\_realloc(), должны быть выровнены на границу в 4 байта.
4. Попытаться уменьшить время поиска свободного блока памяти и время освобождения занятого блока.
5. Попытаться уменьшить фрагментацию памяти.
6. Написать функцию mem\_dump(), которая должна выводить на консоль состояние областей памяти.

## Отчёт

Отчёт должен содержать:

1. Описание разработанного алгоритма.
2. Оценку времени поиска свободного блока памяти, оценку времени освобождения занятого блока.
3. Оценку расхода памяти для хранения служебной информации.
4. Описание достоинств и недостатков разработанного аллокатора.
5. Листинг аллокатора памяти общего назначения.
6. Пример работы аллокатора.
7. **Описание разработанного алгоритма.**

Алгоритм программы был целенаправлен на ускорение поиска свободного фрагмента при выделении памяти функцией *mem\_alloc*, а также на уменьшение фрагментации памяти. Обе цели достигнуты следующим образом. При создании объекта класса «*MemoryAllocator*» ему даётся в распоряжение область памяти некторого размера *TOTAL\_SIZE*. Из этой памяти аллокатором отводится фрагмент размера *TABLE\_SIZE* на создание и хранения таблицы, содержащей в своих строках ссылки на списки всех свободных блоков памяти. Каждая строка таблицы представляет все блоки памяти, размер которых не больше чем:

12

24

48

…

12\*2N

но больше числа, ассоциирующегося с предыдущей строкой. Минимально возможный размер блока памяти равен 8 + 4 байт, то есть 8 байт – длина заголовка и 4 – минимально возможный размер пользовательской памяти. Если пользователь запрашивает менее 4 байт памяти, то размер выделемой памяти всё равно округляется до 4, поскольку блоки должны находиться по адресах, кратных 4.

При инициализации аллокатора строки таблицы создаются до тех пор, пока число 12\*2N, ассоциирующееся с последней строкой, не покроет область памяти, которая остаётся для создания блоков. Размер этой области равен

TOTAL\_SIZE – TABLE\_SIZE ≤ 12\*2N,

TABLE\_SZIE = 4 \* N,

где N – количетсво строк в таблице.

Содиржимое каждой такой строки – 4-байтный адрес блока памяти, размер которого не более чем число, ассоциирующееся с данной строкой, а также больше, чем число, ассоциирующееся с предыдущей строкой. Каждый такой свободный блок содержит после заголовка ссылку на еще один блок, удовлетворяющий данным условиям для текущей строки. Таким образом, каждая строка таблицы представляет собой ссылку на список блоков памяти, размеры которых находятся в некоторых числовых рамках. Каждый такой список заканчивается ссылкой *NULL*.

Особенностью списков, на которые ссылаются строки таблицы, является их упорядоченность по возрастанию размеров блоков в списке. Таким образом, при поиске выделения свободного фрагмента памяти функцией *mem\_alloc* список просматривается начиная от наименьшего по размеру блока до наибольшего. Выбирается наименьший подходящий по размеру блок памяти, что приводит к меньшей фрагментации. Перед поиском блока в списке предварительно выбирается строка, которая содержит блок искомого размера. Если строка с наименьшим размером, которой покрывет запрашиваемый фрагмент памяти, ссылается на *NULL*, то щитается, что блок памяти выделить нельзя, так как в таблице содержатся ссылки на все свободные блоки памяти.

Такой подход не нуждается в увеличении размеров заголовков, так как ссылки на следующие блоки в списке можно хранить в теле блока, а не в заголовке, поскольку хранимые блоки памяти являются свободными.

≤ 12 bytes

≤ 24 bytes

≤ 2i \* (HEADER\_SZ + 4) bytes

.

.

.

≤ TOTAL\_SIZE ─ TABLE\_SIZE bytes

block

NULL

block

block

**TABLE\_SIZE**

block

block

block

block

block

NULL

NULL

NULL

**TOTAL\_SIZE**

**N**

При освобождении блока памяти функцией *mem\_free* освобождаемый блок сливается со свободными соседними. Ссылка на результирующий блок отправляется в таблицу, не нарушая упорядоченности списка блоков по возрастанию размеров блоков.

Оценим расход памяти для хранения служебной информации. В худшем случае

TOTAL\_SIZE ≤ 12\*2N

Найдём количество строк таблицы для заданной области памяти:

В таком случае максимальный размер таблицы в байтах равен

Из данного принципа работы алгоритма следует, что процент расхода памяти на служебную информацию не будет превышать

Время поиска свободного блока памяти сводится к поиску блока в списке заданной строки таблицы, так как поиск строки сводится к вычислению её номера. Поскольку блоки в списке упорядочены по возрастанию их размеров, то можно воспользоваться бинарным поиском. В таком случаем оценка поиска свободного блока будет составлять

,

где K – количество блоков в списке.

В худшем случае в одном списке будут находиться все совбодные блоки, однако такая ситуация возможна только тогда, когда все свободные блоки имеют размер одинавого порядка (одной и той же степени двойки). Стоит отметить, что даже в таком случае поиск не будет осуществляться среди занятых фрагментов, а только лишь среди свободных.

Оценка времени освобождения заданного блока аналогична. В отличие от тривиального освобождения, заданный алгоритм требует добавления блока в список с сохранением упорядоченности, что замедляет функцию в целом.

Преимущества разработанного алгоритма:

* быстрый поиск свободного блока памяти;
* низкая фрагментация памяти;
* небольшой размер заголовка.

Недостатки:

* часть области памяти отводится для таблицы;
* незначительно замедлено время освобождения блока памяти.**3. Текст программы**

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* General-purpose Memory Allocator

\* System software Lab №1 (1st semester 2009)

\*

\* Author:

\* Podolsky Sergey

\* Group: KV-64

\*

\* written: 15.10.2009

\*

\* Project definition:

\* Main.cpp Defines the entry point for the console application

\* MemoryAllocator.h Defines Memory Allocator class

\* stdafx.h Include file for standard system include files

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

#pragma once

#include "stdafx.h"

/\*---------------- Definitions -------------------------\*/

#define BLOCK\_T void\*

#define SIZE\_T unsigned int

#define SIZE\_SZ sizeof(SIZE\_T)

#define HEADER\_SZ (2 \* SIZE\_SZ)

#define STATE\_T SIZE\_T

#define FREE 0

#define USED 0x80000000

#define SIZE\_MASK (USED ^ 0xFFFFFFFF)

/\*---------------- MEMORY ALLOCATOR CLASS --------------\*/

class MemoryAllocator

{

private:

/\*----------------- Loacal variables -------------------\*/

BLOCK\_T table;

SIZE\_T table\_length;

char\* lower\_bound;

/\*----------------- Local methods ----------------------\*/

/\* Get block state \*/

STATE\_T block\_state(BLOCK\_T block)

{

return block == NULL ? NULL : ((SIZE\_T\*)block)[0] & USED;

}

/\* Set block state \*/

void set\_block\_state(BLOCK\_T block, STATE\_T state)

{

if ( block != NULL )

if (state == USED)

((SIZE\_T\*)block)[0] |= USED;

else

((SIZE\_T\*)block)[0] &= SIZE\_MASK;

}

/\* Get block that is situated in the same row of the table

and in the same chain as given block \*/

BLOCK\_T similar\_block(BLOCK\_T block)

{

return block == NULL ? NULL : ((BLOCK\_T\*)block)[2];

}

/\* Set to given reference to another block that is situated in the same row

of the table and in the same chain as given block \*/

void set\_similar\_block(BLOCK\_T block, BLOCK\_T similar)

{

if ( block != NULL )

((BLOCK\_T\*)block)[2] = similar;

}

/\* Get block size with header \*/

SIZE\_T block\_size(BLOCK\_T block)

{

return block == NULL ? NULL : ((SIZE\_T\*)block)[0] & SIZE\_MASK;

}

/\* Get size of previous block with header \*/

SIZE\_T prev\_size(BLOCK\_T block)

{

return block == NULL ? NULL : ((SIZE\_T\*)block)[1];

}

/\* Set block size with header to given block \*/

void set\_block\_size(BLOCK\_T block, SIZE\_T size)

{

if ( block == NULL ) return;

((SIZE\_T\*)block)[0] &= USED;

((SIZE\_T\*)block)[0] |= SIZE\_MASK & size;

}

/\* Set size of previous block with header to given block \*/

void set\_prev\_size(BLOCK\_T block, SIZE\_T size)

{

if ( block != NULL )

((SIZE\_T\*)block)[1] = size;

}

/\* Get previus block \*/

BLOCK\_T prev\_block(BLOCK\_T block)

{

return block == NULL || prev\_size(block) == NULL ? NULL : (BLOCK\_T)((char\*)block - prev\_size(block));

}

/\* Get next block \*/

BLOCK\_T next\_block(BLOCK\_T block)

{

return block == NULL || (char\*)block + block\_size(block) >= lower\_bound ? NULL : (BLOCK\_T\*)((char\*)block + block\_size(block));

}

/\* Get max size of any block that can be represented by specified row of the table \*/

SIZE\_T row\_size(SIZE\_T row)

{

return (HEADER\_SZ + SIZE\_SZ) << row;

}

/\* Add free block to the table and put it automatically

to required row in order of block ascending by size \*/

void add\_block\_to\_table(BLOCK\_T block)

{

if ( block != NULL )

// For each row in table

for ( SIZE\_T row = 0; row < table\_length; row++ )

// If row satisfies size

if ( block\_size(block) <= row\_size(row) )

// For each block in chain

for ( BLOCK\_T current = ((BLOCK\_T\*)table)[row], \*prev = NULL; ; prev = current, current = similar\_block(current) )

if ( current == NULL || block\_size(block) <= block\_size(current) )

{

set\_similar\_block(block, current);

if ( prev == NULL )

((BLOCK\_T\*)table)[row] = block;

else

set\_similar\_block(prev, block);

return;

}

}

/\* Remove specified block from the table for making it used \*/

void remove\_from\_table(BLOCK\_T block)

{

if ( block != NULL )

// For each row in table

for ( SIZE\_T row = 0; row < table\_length; row++ )

// If row satisfies size

if ( block\_size(block) <= row\_size(row) )

// For each block in chain

for ( BLOCK\_T current = ((BLOCK\_T\*)table)[row], \*prev = NULL; ; prev = current, current = similar\_block(current) )

if ( current == block )

{

if ( prev == NULL )

((BLOCK\_T\*)table)[row] = similar\_block(current);

else

set\_similar\_block(prev, similar\_block(current));

return;

}

else if ( current == NULL )

return;

}

/\* Copy specified count of chars from "source" to "dest" \*/

void migrate\_data(char\* source, char\* dest, SIZE\_T offset, SIZE\_T count)

{

for (; offset < count; offset++)

\*(dest + offset) = \*(source + offset);

}

/\*----------------- Public methods ---------------------\*/

public:

/\* Constructor \*/

MemoryAllocator(BLOCK\_T memory\_space, SIZE\_T memory\_size)

{

/\* Assign lower bound\*/

lower\_bound = (char\*)memory\_space + memory\_size;

/\* Create table \*/

table = memory\_space;

// Get table length

table\_length = 0;

do

((BLOCK\_T\*)table)[table\_length] = NULL; // Initialize each table row with NULL

while ( row\_size(table\_length++) < memory\_size - table\_length \* sizeof(BLOCK\_T\*) );

/\* Create one block from all the memory given left \*/

BLOCK\_T block = ((BLOCK\_T\*)table + table\_length);

// Set to last table row value of main block offset

((BLOCK\_T\*)table)[table\_length - 1] = block;

// Initialize main block

set\_block\_size(block, memory\_size - table\_length \* sizeof(BLOCK\_T\*));

set\_prev\_size(block, NULL);

set\_block\_state(block, FREE);

set\_similar\_block(block, NULL);

}

/\* Allocate memory \*/

BLOCK\_T mem\_alloc(SIZE\_T size)

{

/\* Include header to size and increase to multiple of SIZE\_SZ \*/

size += size % SIZE\_SZ == 0 ? HEADER\_SZ : HEADER\_SZ + SIZE\_SZ - size % SIZE\_SZ;

/\* For each table row \*/

for ( SIZE\_T row = 0; row < table\_length; row++)

/\* If current row satisfies size \*/

if ( size <= row\_size(row) )

/\* For each block in chain \*/

for ( BLOCK\_T block = ((BLOCK\_T\*)table)[row], \*prev\_similar = NULL; block != NULL; prev\_similar = block, block = similar\_block(block) )

/\* If block has enough size \*/

if ( block\_size(block) >= size )

{

/\* Set block state to used \*/

set\_block\_state(block, USED);

/\* Remove block from chain \*/

if (prev\_similar == NULL)

((BLOCK\_T\*)table)[row] = similar\_block(block); // Block is first in chain

else

set\_similar\_block(prev\_similar, similar\_block(block)); // Block not first in chain

/\* If block has enough free space to create new block header \*/

if ( block\_size(block) - size > HEADER\_SZ )

{

/\* Truncate current block \*/

SIZE\_T old\_size = block\_size(block);

set\_block\_size(block, size);

/\* Insert and define additional new block \*/

BLOCK\_T inserted\_block = next\_block(block);

set\_block\_size(inserted\_block, old\_size - block\_size(block));

set\_prev\_size(inserted\_block, block\_size(block));

set\_block\_state(inserted\_block, FREE);

BLOCK\_T next = next\_block(inserted\_block);

/\* If inserted block is not last \*/

if ( next != NULL )

/\* Change property of next block that defines previous block's size \*/

set\_prev\_size(next, block\_size(inserted\_block));

/\* Add inserted block to table \*/

add\_block\_to\_table(inserted\_block);

}

return (BLOCK\_T)((char\*)block + HEADER\_SZ); /\* Return block memory offset \*/

}

return NULL; // Free block of requested size is not found

}

/\* Free memory \*/

void mem\_free(BLOCK\_T addr)

{

if ( addr == NULL ) return;

/\* Get block offset \*/

BLOCK\_T block = (BLOCK\_T)((char\*)addr - HEADER\_SZ);

/\* Get previous and next blocks \*/

BLOCK\_T prev = prev\_block(block);

BLOCK\_T next = next\_block(block);

/\* Merge current block with both next and previous \*/

if ( (next != NULL && block\_state(next) == FREE) && (prev != NULL && block\_state(prev) == FREE) )

{

remove\_from\_table(prev);

remove\_from\_table(next);

set\_block\_size(prev, block\_size(prev) + block\_size(block) + block\_size(next));

next = next\_block(prev);

if (next != NULL)

set\_prev\_size(next, block\_size(prev));

add\_block\_to\_table(prev);

}

/\* Merge current block with previous \*/

else if ( prev != NULL && block\_state(prev) == FREE )

{

remove\_from\_table(prev);

set\_block\_size(prev, block\_size(prev) + block\_size(block));

if ( next != NULL)

set\_prev\_size(next, block\_size(prev));

add\_block\_to\_table(prev);

}

/\* Merge current block with next \*/

else if ( next != NULL && block\_state(next) == FREE )

{

remove\_from\_table(next);

set\_block\_state(block, FREE);

set\_block\_size(block, block\_size(block) + block\_size(next));

next = next\_block(block);

if (next != NULL)

set\_prev\_size(next, block\_size(block));

add\_block\_to\_table(block);

}

/\* Set current block as free without merging \*/

else

{

set\_block\_state(block, FREE);

add\_block\_to\_table(block);

}

}

/\* Reallocate memory \*/

BLOCK\_T mem\_realloc(BLOCK\_T addr, SIZE\_T size)

{

if ( addr == NULL ) return mem\_alloc(size);

/\* Get block pointer \*/

BLOCK\_T block = (BLOCK\_T)((char\*)addr - HEADER\_SZ);

/\* The same size requested \*/

if ( size + (size % SIZE\_SZ == 0 ? HEADER\_SZ : HEADER\_SZ + SIZE\_SZ - size % SIZE\_SZ) == block\_size(block) ) return addr;

/\* Try to allocate new memory with user-defined new size \*/

BLOCK\_T new\_addr = mem\_alloc(size);

/\* Allocated successfully \*/

if ( new\_addr != NULL )

{

migrate\_data((char\*)addr, (char\*)new\_addr, 0, size);

mem\_free(addr);

}

/\* If coluld not allocate \*/

else

{

/\* Get current, previous and next blocks \*/

BLOCK\_T next = next\_block(block), \*prev = prev\_block(block);

/\* Get posible size we can receive after free memory \*/

SIZE\_T posible\_size =

(next != NULL && block\_state(next) == FREE) && (prev != NULL && block\_state(prev) == FREE) ? block\_size(prev) + block\_size(block) + block\_size(next) :

(prev != NULL && block\_state(prev) == FREE) ? block\_size(prev) + block\_size(block) :

(next != NULL && block\_state(next) == FREE) ? block\_size(next) + block\_size(block) :

block\_size(block);

/\* If possible free memory is too small then return NULL \*/

if ( posible\_size - HEADER\_SZ < size ) return NULL;

/\* Possible free size would be enough \*/

BLOCK\_T data = similar\_block(block); // Get first several bytes of data in block

mem\_free(addr); /\* Free current block \*/

new\_addr = mem\_alloc(size); /\* Allocate memory \*/

set\_similar\_block((BLOCK\_T)((char\*)new\_addr - HEADER\_SZ), data); // Get first several bytes of data in block

if ( new\_addr == addr ) return addr; /\* The same block returned \*/

migrate\_data((char\*)addr, (char\*)new\_addr, sizeof(BLOCK\_T), size - sizeof(BLOCK\_T));

}

return new\_addr;

}

/\* Dump memory state to console \*/

void mem\_dump()

{

SIZE\_T count = 0;

for ( BLOCK\_T block = (BLOCK\_T\*)table + table\_length; block != NULL; block = next\_block(block), count++ )

cout << block << ":" /\* Offset \*/

<< "\tSize: " << HEADER\_SZ << "+" << dec << block\_size(block) - HEADER\_SZ /\* Header + size\*/

<< "\tState: " << (block\_state(block) == FREE ? "FREE" : "USED") << endl; /\* USED/FREE \*/

//cout << "Total block count:\t" << count << endl;

cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << endl;

}

/\* Dump table to console \*/

void table\_dump()

{

for ( SIZE\_T row = 0; row < table\_length; row++)

{

SIZE\_T count = 0;

for ( BLOCK\_T block = ((BLOCK\_T\*)table)[row]; block != NULL; block = similar\_block(block) )

count++;

cout << dec << row << "\t(" << row\_size(row) << "):\t" << count << endl;

}

}

};

**4. Пример работы аллокатора**

unsigned size = 1024;

MemoryAllocator \*allocator = new MemoryAllocator(malloc(size), size);

allocator->table\_dump();

allocator->mem\_dump();

void \*addr1 = allocator->mem\_alloc(100);

void \*addr2 = allocator->mem\_alloc(100);

void \*addr3 = allocator->mem\_alloc(100);

allocator->table\_dump();

allocator->mem\_dump();

addr2 = allocator->mem\_realloc(addr2, 120);

allocator->table\_dump();

allocator->mem\_dump();

allocator->mem\_free(addr3);

allocator->table\_dump();

allocator->mem\_dump();

allocator->mem\_free(addr2);

allocator->table\_dump();

allocator->mem\_dump();

**CONSOLE OUTPUT:**

0 (12): 0

1 (24): 0

2 (48): 0

3 (96): 0

4 (192): 0

5 (384): 0

6 (768): 0

7 (1536): 1

00328FB8: Size: 8+984 State: FREE

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

0 (12): 0

1 (24): 0

2 (48): 0

3 (96): 0

4 (192): 0

5 (384): 0

6 (768): 1

7 (1536): 0

00328FB8: Size: 8+100 State: USED

00329024: Size: 8+100 State: USED

00329090: Size: 8+100 State: USED

003290FC: Size: 8+660 State: FREE

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

0 (12): 0

1 (24): 0

2 (48): 0

3 (96): 0

4 (192): 1

5 (384): 0

6 (768): 1

7 (1536): 0

00328FB8: Size: 8+100 State: USED

00329024: Size: 8+100 State: FREE

00329090: Size: 8+100 State: USED

003290FC: Size: 8+120 State: USED

0032917C: Size: 8+532 State: FREE

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

0 (12): 0

1 (24): 0

2 (48): 0

3 (96): 0

4 (192): 0

5 (384): 1

6 (768): 1

7 (1536): 0

00328FB8: Size: 8+100 State: USED

00329024: Size: 8+208 State: FREE

003290FC: Size: 8+120 State: USED

0032917C: Size: 8+532 State: FREE

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

0 (12): 0

1 (24): 0

2 (48): 0

3 (96): 0

4 (192): 0

5 (384): 0

6 (768): 0

7 (1536): 1

00328FB8: Size: 8+100 State: USED

00329024: Size: 8+876 State: FREE

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Press any key to continue . . .