**Dynamic Memory Allocator**

**ADVANCES IN OPERATING SYSTEM**

2023 - 24

**ABSTRACT**

The Dynamic memory allocation is crucial for efficient memory management in modern programming. This project involves designing and implementing a custom dynamic memory allocator in C++, which allocates the standard memory and manages it just like built-in memory management functions like malloc() and free().

The custom allocator provides a comprehensive approach to managing memory by handling alignment, block management, and interaction with the operating system. The implementation leverages fundamental system calls such as sbrk() to request and release memory from the operating system.

This report outlines the architecture, design considerations, and implementation details of the allocator. It also includes practical test cases to validate the allocator's functionality, demonstrating its capability to handle various allocation scenarios effectively. This custom allocator serves as a valuable educational tool for understanding the inner workings of dynamic memory management.

i

Table of Contents

ABSTRACT i

[CHAPTER 1: INTRODUCTION 1](#_TOC_250040)

* 1. [OVERVIEW 5](#_TOC_250038)
  2. [OBJECTIVES 5](#_TOC_250037)
  3. [SCOPE 6](#_TOC_250036)

[CHAPTER 2: BACKGROUND 7](#_TOC_250034)

* 1. [MEMORY ALLOCATION 7](#_TOC_250032)
  2. MEMORY LAYOUT OF PROGRAM 8

[CHAPTER 3: IMPLEMENTATION 10](#_TOC_250020)

* 1. [STRUCTURE OF THE MEMORY BLOCK 10](#_TOC_250018)
  2. [MEMORY ALIGNMENT 11](#_TOC_250017)
  3. MEMORY MAPPING 11
  4. FREEING OBJECTS 12
  5. RESUSING BLOCKS 12

[CHAPTER 4: EVALUATION 13](#_TOC_250016)

* 1. [ALIGNMENT TEST 13](#_TOC_250014)
  2. [ALLOCATION TEST 13](#_TOC_250013)
  3. [FREEING MEMORY TEST 13](#_TOC_250012)
  4. REUSING MEMORY TEST 13

[CHAPTER 5: RESULTS 14](#_TOC_250011)

[CHAPTER 6: CONCLUSION 15](#_TOC_250006)

**Chapter 1: INTRODUCTION**

The Overview of the Project, the Problem Statement, Objectives, and the Basic Methodology or the Flow of the model is defined and discussed in this chapter. This is helpful to further discuss the concepts behind the model and project in depth.

### OVERVIEW

Dynamic memory allocation allows programs to request memory during execution, providing flexibility in memory usage and enabling efficient resource management. Unlike static and automatic memory allocation, dynamic allocation occurs at runtime, allowing programs to adapt to varying memory needs. This project focuses on creating a custom dynamic memory allocator to gain a deeper understanding of how dynamic memory is managed and to explore the underlying mechanisms of standard allocation functions.

### 

### OBJECTIVES

The primary objectives of this project are:

1. **Implementation of a Custom Allocator**: Develop a memory allocator in C++ that mimics the behavior of standard allocation functions like malloc() and free().
2. **Memory Management Techniques**: Implement key memory management techniques such as alignment, block management, and memory mapping.
3. **Interaction with the Operating System**: Use system calls to request and manage memory from the operating system, specifically utilizing sbrk() to adjust the heap size.
4. **Validation and Testing**: Ensure the correctness of the allocator through rigorous testing, including alignment checks, exact allocation, freeing of memory, and block reuse.

### SCOPE

The project encompasses the following aspects:

1. **Design and Implementation**: Create a memory block structure to manage allocated memory, implement allocation and deallocation functions, and handle memory alignment.
2. **System Interaction**: Use system calls to request additional memory from the operating system and manage the heap.
3. **Testing and Valida**tion: Develop test cases to verify the functionality and correctness of the custom allocator, including alignment, allocation, and block reuse.

## Chapter 2: BACKGROUND

Understanding dynamic memory allocation requires a grasp of the fundamental concepts and memory layout of C programs. This section provides an overview of memory allocation types and the memory layout of a C program.

**2.1 Memory Allocation in C**

**Static Allocation:**

Memory is allocated at the program's startup and remains throughout the program's execution. Global and static variables are examples of static allocation. The memory is reserved during compile time and persists for the program's lifetime.

Example: int globalVar = 5;

**Automatic Allocation:**

Memory is allocated when entering a block (e.g., a function) and deallocated when exiting the block. Local variables and function arguments are allocated on the stack, which is managed automatically.

Example: void func() { int localVar = 10; }

**Dynamic Allocation:**

Memory is allocated and managed at runtime using functions like malloc(), calloc(), realloc(), and free(). The programmer controls the allocation and deallocation of memory, allowing for flexible memory management.

Example: int \*ptr = (int \*)malloc(sizeof(int));

**2.2 Memory Layout of a C Program**

After compilation of a program, binary executable file or output file is created, which when executed loads into RAM in an organized manner. Memory layout in C Program has six components which are text segment, initialized data segment, uninitialized data segment, command-line arguments, stack, and heap.

Each of these segments stores different parts of code and have their own read, write permissions. If a program tries to access the value stored in any segment differently than it is supposed to, it results in a segmentation fault error.

Upon compilation and execution, a C program's memory is organized into several segments:

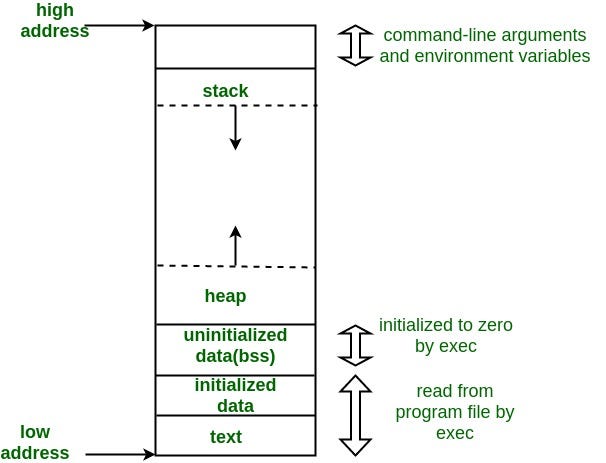


Figure 2 1: Memory Layout of Program

**1. Text Segment:**

- Description: Contains the executable code of the program.

- Permissions: Typically read-only and executable. This segment is shared among processes.

- Example: The compiled machine code of the program.

**2. Initialized Data Segment:**

- Description: Stores global and static variables that are initialized by the programmer.

- Permissions: Read-write for global/static variables and read-only for constants.

- Example: `int globalVar = 10;`

**3. Uninitialized Data Segment (BSS):**

- Description: Contains global and static variables that are uninitialized.

- Permissions: Read-write. The operating system initializes these variables to zero.

- Example: `static int uninitializedVar;`

**4. Stack:**

- Description: Manages function calls, local variables, and return addresses.

- Characteristics: Operates on a Last In, First Out (LIFO) principle.

- Each function call creates a stack frame.

- Example: Local variables and function parameters.

**5. Heap:**

- Description: Manages dynamically allocated memory during program execution.

- Characteristics: Grows and shrinks based on allocation and deallocation.

- Managed by functions like `malloc()` and `free()`.

- Example: Memory allocated with `malloc()`.

**6. Command Line Arguments:**

- Description: Stores command-line parameters passed to the program.

- Characteristics: `argc` holds the number of arguments, and `argv` is an array of arguments.

- Example: `int main(int argc, char \*argv[])`

### 

### Chapter 3: IMPLEMENTATION

This section details the design and implementation of the custom dynamic memory allocator, including the structure of memory blocks, memory alignment, mapping, freeing objects, and block reuse.

**3.1 Structure of the Memory Block**

The memory block structure is crucial for managing allocated memory:

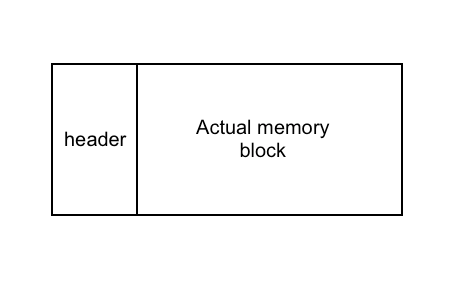


Figure 3 1: Structure of Memory Block

struct Block {

size\_t size; // Size of the block, including metadata and user data

bool used; // Status indicating if the block is in use

Block \*next; // Pointer to the next block in the linked list

word\_t data[1]; // Flexible array member for user data

};

- `size`: Represents the total size of the block, including metadata and data.

- `used`: Indicates whether the block is currently allocated.

- `next`: Pointer to the next block in the linked list for traversal.

- `data`: A flexible array member used to store user data.

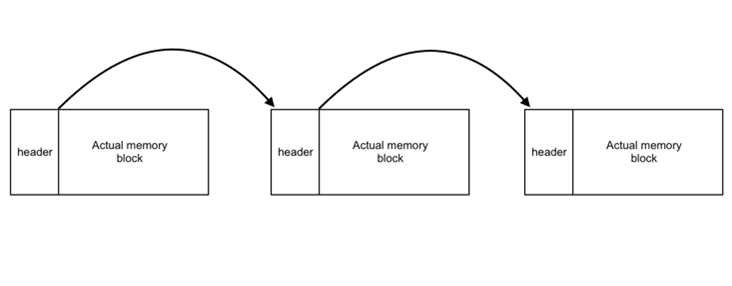


Figure 3 2: Contiguous Memory Blocks

**3.2 Memory Alignment**

Memory alignment ensures efficient access to memory by aligning addresses to the machine word boundary. This is achieved using the `align()` function:

inline size\_t align(size\_t n) {

return (n + sizeof(word\_t) - 1) & ~(sizeof(word\_t) - 1);

}

Purpose: Aligns the requested size to the nearest multiple of the machine word size.

Example: For a 4-byte word size, requesting 6 bytes will be aligned to 8 bytes.

**3.3 Memory Mapping**

Memory mapping involves requesting and managing memory from the operating system. The `sbrk()` system call is used to adjust the heap size:

Block \*requestFromOS(size\_t size) {

auto block = (Block \*)sbrk(0);

if (sbrk(allocSize(size)) == (void \*)-1)

return nullptr;

return block;

}

- `sbrk(0)`: Returns the current end of the heap.

- `sbrk(size)`: Requests additional memory by adjusting the heap size.

- `allocSize()`: Calculates the total size needed, including block metadata.

**3.4 Freeing Objects**

The `free()` function marks a block as unused, allowing it to be reused:

void free(word\_t \*data) {

auto block = getHeader(data);

block->used = false;

}

- `getHeader()`: Retrieves the block header from the user data pointer.

- `used`: Sets the `used` flag to `false` to indicate the block is free.

**3.5 Reusing Blocks**

Reusing freed blocks helps reduce fragmentation and improves efficiency:

Block \*findFreeBlock(size\_t size) {

auto block = heapStart;

while (block) {

if (!block->used && block->size >= size)

return block;

block = block->next;

}

return nullptr;

}

- `findFreeBlock()`: Searches the linked list for a free block that meets the requested size.

- Reuse: If a suitable block is found, it is reused for the allocation request

### Chapter 4: EVALUATION

Testing and evaluation ensure the custom allocator functions as expected. This section includes detailed test cases and their results. The following test cases verify the allocator's functionality:

**Alignment Test:**

Objective: Ensure that memory is aligned correctly.

Method: Allocate 3 bytes and verify that the size is aligned to the nearest machine word boundary.

Expected Result: The allocated size should match the alignment requirements. For a 4-byte machine word, the size should be rounded up to 4 bytes.

**Exact Allocation Test:**

Objective: Verify that the allocator can handle exact size requests.

Method: Allocate 8 bytes and ensure the allocated size matches the requested size.

Expected Result: The allocated block should have the correct size of 8 bytes.

**Freeing Memory Test:**

Objective: Ensure that memory blocks can be freed and marked as available for reuse.

Method: Allocate 8 bytes, free the block, and verify that the block is marked as unused.

Expected Result: The block should be marked as unused and should be available for future allocations.

**Reusing Memory Test:**

Objective: Test if a freed block is reused correctly.

Method: Allocate 8 bytes, free the block, and then allocate 8 bytes again to check the reuse.

Expected Result: The newly allocated block should be the same as the previously freed block.

**Chapter 5: RESULTS**

The tests were conducted using the custom allocator. The results were as follows:

**Alignment Test:**

Result: Successful. The size of allocated memory was correctly aligned to the nearest machine word boundary, as verified by the assertion.

**Exact Allocation Test:**

Result: Successful. The allocated memory size matched the requested size, confirming that exact allocation is handled properly.

**Freeing Memory Test:**

Result: Successful. The block was correctly marked as unused, and the system was able to identify and reuse this block in subsequent allocations.

**Reusing Memory Test:**

Result: Successful. The allocator successfully reused the previously freed block, demonstrating that the allocator can manage memory efficiently and reduce fragmentation.

The tests confirm that the custom allocator handles memory alignment, exact allocation, freeing of memory, and block reuse effectively. The implementation demonstrates correct interaction with the operating system's memory management and provides a robust mechanism for dynamic memory allocation.

# 

# CHAPTER 6: CONCLUSION

The custom dynamic memory allocator project provides valuable insights into memory management and allocation techniques. The project successfully implemented a basic memory allocator that mimics the behavior of standard functions such as malloc() and free(). Key aspects of the implementation included memory alignment, block management, and interaction with the operating system through the sbrk() system call.

Summary:

* Developed a custom memory allocator capable of handling dynamic memory requests.
* Implemented alignment, block management, and efficient memory reuse.
* Utilized system calls to request and manage memory from the operating system.
* Thorough testing validated the correctness and functionality of the allocator.

This project has provided a solid foundation in understanding dynamic memory management and the intricacies of custom allocator implementation. The knowledge gained is instrumental for developing more advanced memory management systems and for a deeper appreciation of the mechanisms underlying standard allocation functions.