Memory Management System Documentation

BT21CSE015 Prajwal Sam

January 23, 2024

CSL 316 - Assignment 1- Semester 6 2024
Progammer: Prajwal Sam Rachapudy
Date Due: 24th Jan 2024

Purpose: This C++ program implements a simple memory manager that handles memory allocation, deallocation, and compaction. It uses a list of memory blocks to manage both used and free memory spaces.
The memory manager reads commands from an input file to allocate, deallocate, and assign memory.

Overview

The provided C++ code implements a simple memory management system, allowing dynamic allocation and deallocation of memory blocks. The memory manager maintains information about used and free memory blocks, facilitating efficient allocation and deallocation operations. Additionally, the system supports memory compaction to optimize the use of available memory.

Assumptions

1. The memory manager is designed for educational purposes and may not be suitable for production environments.

2. The maximum memory size is predefined using the MEMORY SIZE _MB macro (default is 64 MB).

3. The system assumes non-negative sizes for memory allocations.

4. The input file format is assumed to follow a specific pattern (see Sample Input section).

Memory Block Structure

The MemoryBlock struct contains information about a memory block, including the variable name, start address, size, and reference count.

```
struct MemoryBlock
{
    std::string var_name;
    int start_address;
    int size;
    int reference_count;
};
```

Memory Manager Structure

The MemoryManager struct manages both used and free memory blocks.

```
struct MemoryManager
{
    std::vector<MemoryBlock> used_blocks;
    std::vector<MemoryBlock> free_blocks;
};
```

Functions

1. Create memory manager()

This function initializes a memory manager with one free block representing the entire available memory.

```
MemoryManager create_memory_manager()
{
    MemoryManager manager;
    manager.free_blocks.push_back({"", 0, MEMORY_SIZE, 0});
    return manager;
}
```

2. allocate memory(MemoryManager &manager, int size, std::string name)

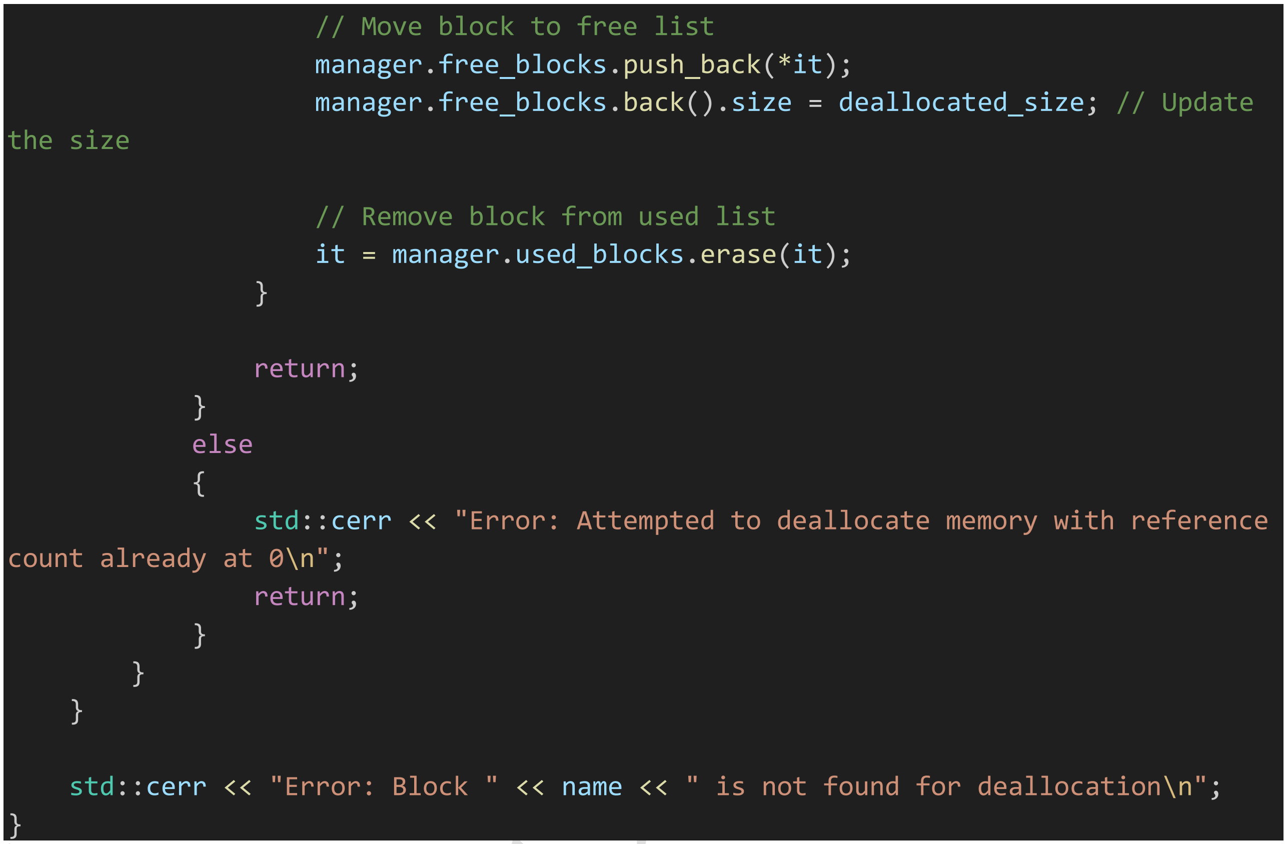
Allocates a memory block of the specified size and associates it with the given variable name.

```
MemoryBlock *allocate_memory(MemoryManager &manager, int size, std::string name)
{
    for (auto it = manager.free_blocks.begin(); it != manager.free_blocks.end();
++it)
    {
        if (it->size >= size)
        {
            MemoryBlock allocated_block;
            allocated_block.var_name = name;
            allocated_block.start_address = it->start_address;
            allocated_block.size = size;
            allocated_block.reference_count = 1;

            // Update free blocks
            int remaining_size = it->size - size;
            if (remaining_size > 0)
            {
                // If there is remaining space, create a new free block
                it->start_address += size;
                it->size = remaining_size;
            }
            else
            {
                // Remove the block from the free list
                it = manager.free_blocks.erase(it);
            }

            // Update used blocks
            manager.used_blocks.push_back(allocated_block);

            return &(manager.used_blocks.back());
        }
    }
    return nullptr; // Allocation failed
}
```

3. deallocate memory(MemoryManager &manager, std::string name)

Deallocates the memory block associated with the given variable name.

```
void deallocate_memory(MemoryManager &manager, std::string name)
{
    for (auto it = manager.used_blocks.begin(); it != manager.used_blocks.end();
++it)
    {
        if (it->var_name == name)
        {
            if (it->reference_count > 0)
            {
                it->reference_count--;

                // Check if reference count becomes 0 after decrementing
                if (it->reference_count == 0)
                {
                    int deallocated_size = it->size;
```

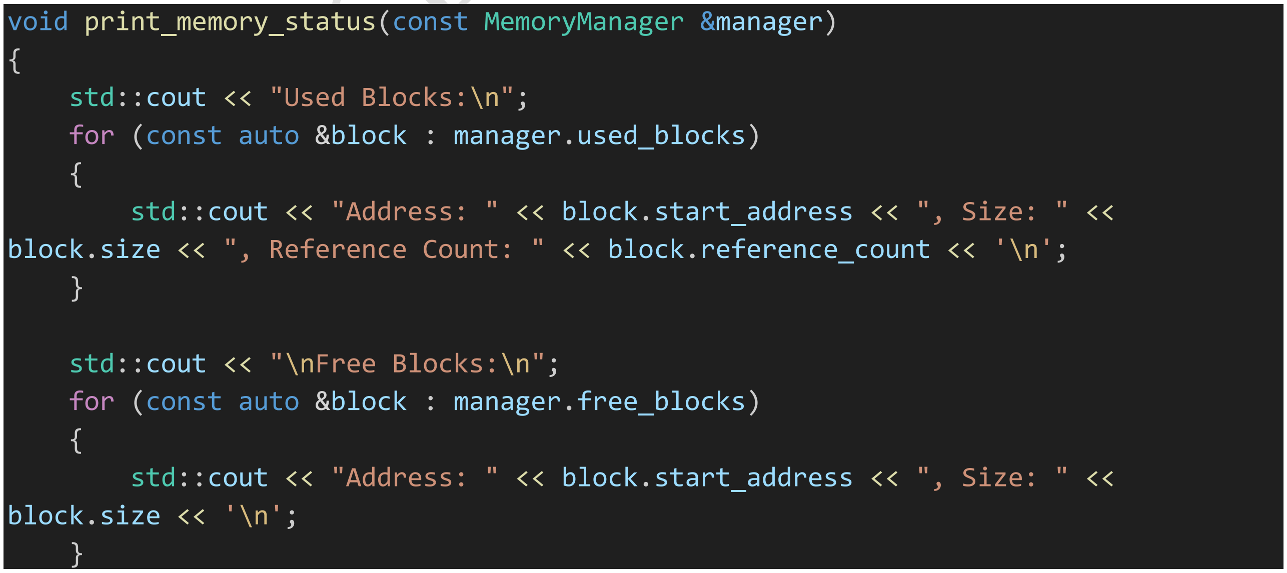


```
                    // Move block to free list
                    manager.free_blocks.push_back(*it);
                    manager.free_blocks.back().size = deallocated_size; // Update
the size

                    // Remove block from used list
                    it = manager.used_blocks.erase(it);
                }

                return;
            }
            else
            {
                std::cerr << "Error: Attempted to deallocate memory with reference
count already at 0\n";
                return;
            }
        }
    }

    std::cerr << "Error: Block " << name << " is not found for deallocation\n";
}
```

1

4. print memory status(const MemoryManager &manager)

Prints the current status of used and free memory blocks.



```
void print_memory_status(const MemoryManager &manager)
{
    std::cout << "Used Blocks:\n";
    for (const auto &block : manager.used_blocks)
    {
        std::cout << "Address: " << block.start_address << ", Size: " <<
block.size << ", Reference Count: " << block.reference_count << '\n';
    }

    std::cout << "\nFree Blocks:\n";
    for (const auto &block : manager.free_blocks)
    {
        std::cout << "Address: " << block.start_address << ", Size: " <<
block.size << '\n';
    }
}
```

1

5. compact memory(MemoryManager &manager)

Compacts the memory by sorting used blocks and updating their start addresses, then recreates free blocks.

```
void compact_memory(MemoryManager &manager)
{
    std::sort(manager.used_blocks.begin(), manager.used_blocks.end(),
              [](const MemoryBlock &a, const MemoryBlock &b)
              {
                  return a.start_address < b.start_address;
              });

    size_t current_address = 0;
    for (auto &block : manager.used_blocks)
    {
        block.start_address = static_cast<int>(current_address);
        current_address += block.size;
    }

    // Update free blocks
    manager.free_blocks.clear(); // Clear existing free blocks

    // Find gaps between used blocks and create new free blocks
    for (size_t i = 0; i < manager.used_blocks.size() - 1; ++i)
    {
        size_t gap_size = manager.used_blocks[i + 1].start_address -
(manager.used_blocks[i].start_address + manager.used_blocks[i].size);
        if (gap_size > 0)
        {
            manager.free_blocks.push_back({"",
static_cast<int>(manager.used_blocks[i].start_address +
manager.used_blocks[i].size), static_cast<int>(gap_size), 0});
        }
    }

    // Add the remaining memory as a free block
    size_t remaining_memory = MEMORY_SIZE - current_address;
    if (remaining_memory > 0)
    {
        manager.free_blocks.push_back({"", static_cast<int>(current_address),
static_cast<int>(remaining_memory), 0});
    }
}
```

6. assign memory(MemoryManager &manager, std::string a, std::string b)

Assigns memory for variable ’a’ as a pointer to the memory block associated with variable ’b’.

```
void assign_memory(MemoryManager &manager, std::string a, std::string b)
{
    // Check if variable 'a' is already declared
    for (const auto &block : manager.used_blocks)
    {
        if (block.var_name == a)
        {
            std::cerr << "Error: Variable " << a << " is already declared\n";
            return;
        }
    }

    // Find the block corresponding to variable 'b'
    for (auto &block : manager.used_blocks)
    {
        if (block.var_name == b)
        {
            // block.reference_count++;
            // Create a new block for variable 'a' as a pointer to 'b'
            MemoryBlock allocated_block;
            allocated_block.var_name = a;
            allocated_block.start_address = block.start_address;
            allocated_block.size = 0;             // Set the size to 0 for a
pointer
            allocated_block.reference_count = 1; // Set reference count to 1 for a
pointer

            // Update used blocks
            manager.used_blocks.push_back(allocated_block);

            std::cout << "Pointer " << a << " to " << b << " is declared\n";
            return;
        }
    }

    std::cerr << "Error: Variable " << b << " not found for assignment\n";
}
```

1

7. main()

The main function reads input commands from a file and performs memory allocation, deallocation, and assignment operations.

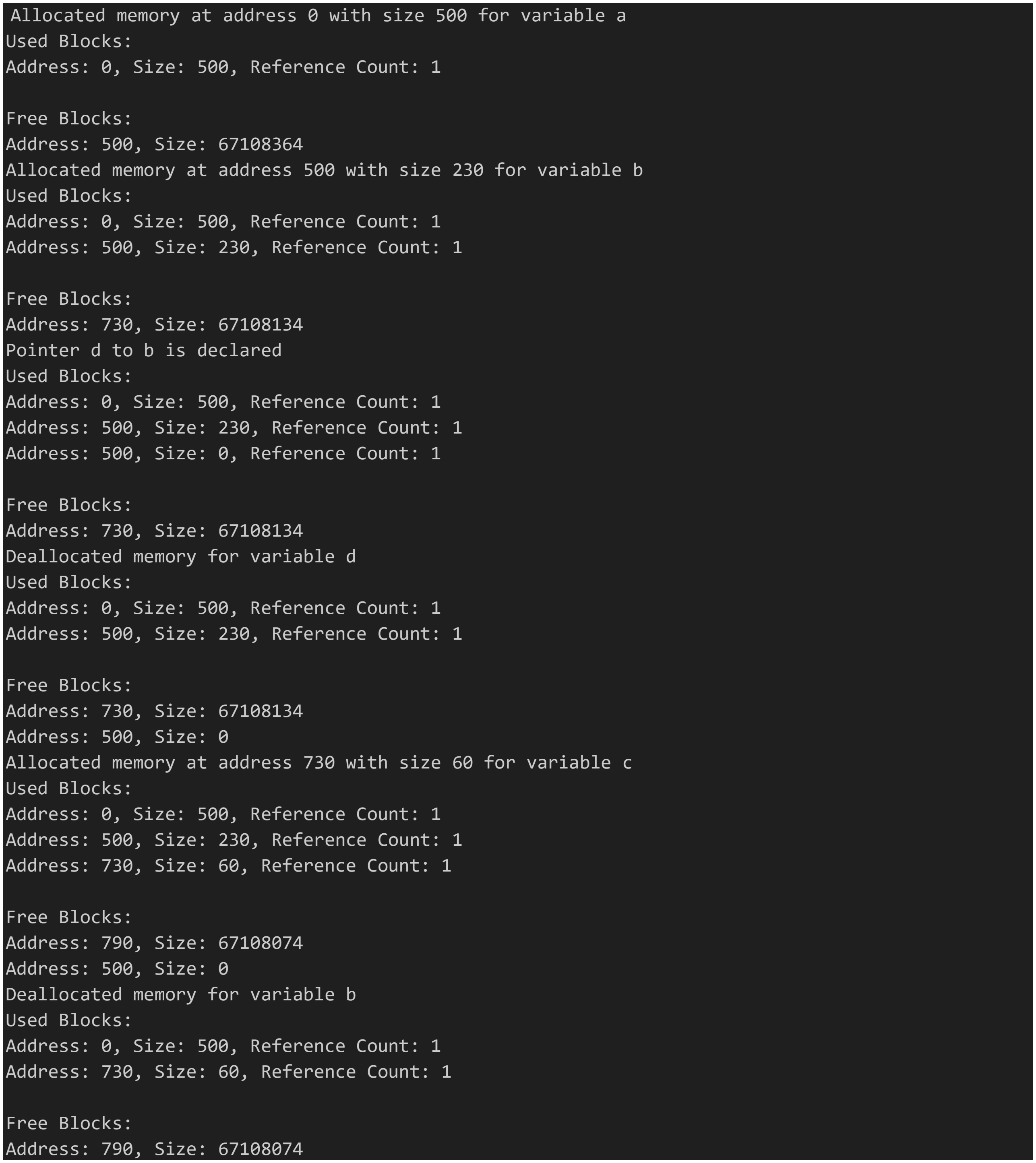
```
int main()
{
    int size;
    MemoryManager memory_manager = create_memory_manager();

    std::ifstream file("input.txt");
    if (!file.is_open())
    {
        std::cerr << "Error opening file\n";
        return 1;
    }

    std::string line;
    while (std::getline(file, line))
    {
        std::istringstream iss(line);
        std::string variable, command, last;

        iss >> variable >> command;

        if (command == "allocate")
        {
            iss >> size;
            MemoryBlock *block = allocate_memory(memory_manager, size, variable);
            if (block)
            {
                std::cout << "Allocated memory at address " << block-
>start_address << " with size " << block->size << " for variable " << variable <<
'\n';
            }
            else
            {
                std::cout << "Error: Insufficient memory for allocation\n";
            }
            print_memory_status(memory_manager);
        }
        else if (command == "=")
        {
            iss >> last;
            assign_memory(memory_manager, variable, last);

            if (memory_manager.used_blocks.empty())
            {
                std::cout << "Error: No block with reference count greater than 0
found for assignment\n";
            }
            print_memory_status(memory_manager);
        }
        else if (command == "free")
        {
            deallocate_memory(memory_manager, variable);
            std::cout << "Deallocated memory for variable " << variable << '\n';
            print_memory_status(memory_manager);
        }
    }

    file.close();
    compact_memory(memory_manager);
    std::cout << "\nMemory status after compacting\n\n";
    print_memory_status(memory_manager);

    return 0;
}
```

Sample Input and Output

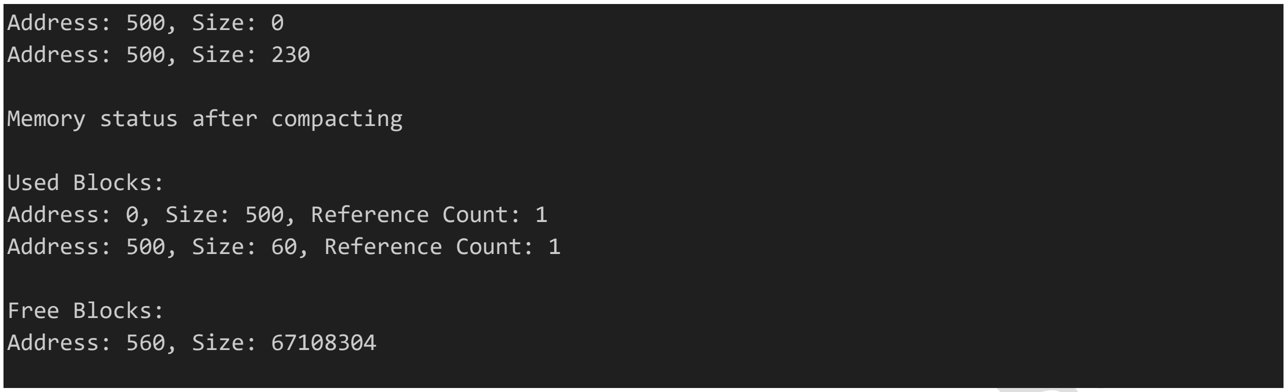
Sample Input (input.txt)

```
a allocate 500
b allocate 230
d = b
d free
c allocate 60
b free
```

Sample Output



```
 Allocated memory at address 0 with size 500 for variable a
Used Blocks:
Address: 0, Size: 500, Reference Count: 1

Free Blocks:
Address: 500, Size: 67108364
Allocated memory at address 500 with size 230 for variable b
Used Blocks:
Address: 0, Size: 500, Reference Count: 1
Address: 500, Size: 230, Reference Count: 1

Free Blocks:
Address: 730, Size: 67108134
Pointer d to b is declared
Used Blocks:
Address: 0, Size: 500, Reference Count: 1
Address: 500, Size: 230, Reference Count: 1
Address: 500, Size: 0, Reference Count: 1

Free Blocks:
Address: 730, Size: 67108134
Deallocated memory for variable d
Used Blocks:
Address: 0, Size: 500, Reference Count: 1
Address: 500, Size: 230, Reference Count: 1

Free Blocks:
Address: 730, Size: 67108134
Address: 500, Size: 0
Allocated memory at address 730 with size 60 for variable c
Used Blocks:
Address: 0, Size: 500, Reference Count: 1
Address: 500, Size: 230, Reference Count: 1
Address: 730, Size: 60, Reference Count: 1

Free Blocks:
Address: 790, Size: 67108074
Address: 500, Size: 0
Deallocated memory for variable b
Used Blocks:
Address: 0, Size: 500, Reference Count: 1
Address: 730, Size: 60, Reference Count: 1

Free Blocks:
Address: 790, Size: 67108074
```



```
Address: 500, Size: 0
Address: 500, Size: 230

Memory status after compacting

Used Blocks:
Address: 0, Size: 500, Reference Count: 1
Address: 500, Size: 60, Reference Count: 1

Free Blocks:
Address: 560, Size: 67108304
```