LP: Compilers Assignment 1

C++ MEMORY MANAGEMENT SIMULATION

CSL316 - ASSIGNMENT 1 - 6TH SEMESTER, 2024

NAME: RAVI BISHNOI
ENROLLMENT NO: BT21CSE117
TA: SHRAVAN KUMAR PURVE SIR

Purpose:

This program simulates memory management as done in interpreted languages like python or java. It allocates and deallocates memory, maintains reference counts, and performs memory compaction. The program reads transactions from an input file and prints the contents of memory structures.

Ravi Bishnoi | CSL316 | 21/01/2024

https://github.com/Doctor9Trio/Memory_management_simulation

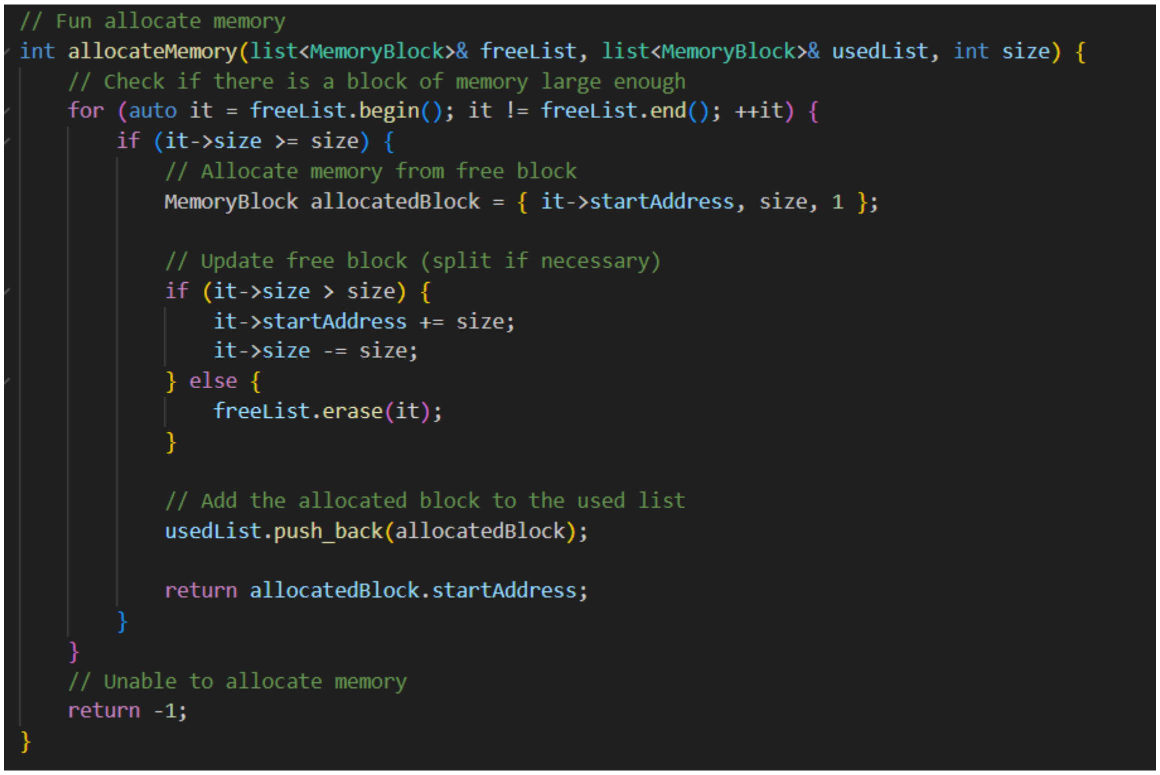
Code documentation:

```
#include <iostream>

#include <list>

#include <unordered_map>

#include <algorithm>

#include <fstream>


using namespace std;


// Define a block of memory

struct MemoryBlock {

int startAddress;

int size;

int referenceCount;

};


// Function prototypes

int allocateMemory(list<MemoryBlock>&freeList, list<MemoryBlock>&usedList,
int size);

void deallocateMemory(list<MemoryBlock>&freeList,
list<MemoryBlock>&usedList, int startAddress);
```

void compactMemory(list<MemoryBlock>&freeList, list<MemoryBlock>&usedList);

void printMemory(list<MemoryBlock>&freeList, list<MemoryBlock>&usedList);

- **Function:allocateMemory()**



```
// Fun allocate memory
int allocateMemory(list<MemoryBlock>& freeList, list<MemoryBlock>& usedList, int size) {
    // Check if there is a block of memory large enough
    for (auto it = freeList.begin(); it != freeList.end(); ++it) {
        if (it->size >= size) {
            // Allocate memory from free block
            MemoryBlock allocatedBlock = { it->startAddress, size, 1 };

            // Update free block (split if necessary)
            if (it->size > size) {
                it->startAddress += size;
                it->size -= size;
            } else {
                freeList.erase(it);
            }

            // Add the allocated block to the used list
            usedList.push_back(allocatedBlock);

            return allocatedBlock.startAddress;
        }
    }
    // Unable to allocate memory
    return -1;
}
```

Use:

Allocates a block of memory from the free list.

Arguments:

- freeList: a list of free memory blocks (modified by allocateMemory).

- usedList: a list of used memory blocks (modified by allocateMemory).

- size: the size of the memory block to allocate.

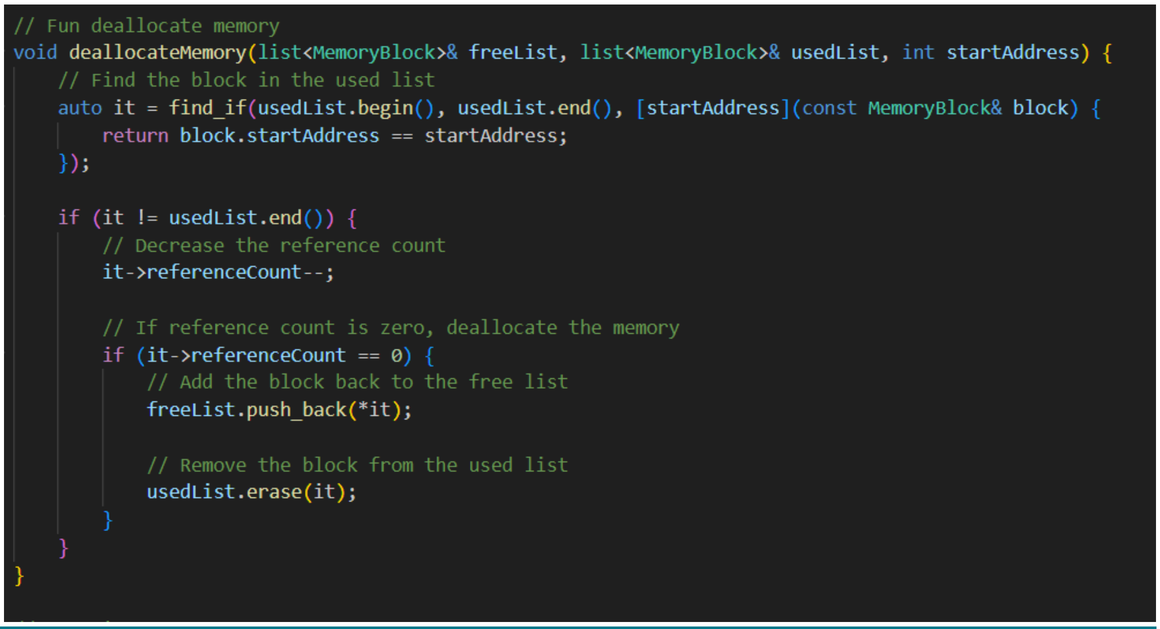
Returns:

The starting address of the allocated block, or -1 if allocation fails.

Notes:

If the free block is larger than needed, it may be split.

- ## Function: deallocateMemory()



```cpp
// Fun deallocate memory
void deallocateMemory(list<MemoryBlock>& freeList, list<MemoryBlock>& usedList, int startAddress) {
    // Find the block in the used list
    auto it = find_if(usedList.begin(), usedList.end(), [startAddress](const MemoryBlock& block) {
        return block.startAddress == startAddress;
    });

    if (it != usedList.end()) {
        // Decrease the reference count
        it->referenceCount--;

        // If reference count is zero, deallocate the memory
        if (it->referenceCount == 0) {
            // Add the block back to the free list
            freeList.push_back(*it);

            // Remove the block from the used list
            usedList.erase(it);
        }
    }
}
```

Use:

Decreases the reference count and deallocates memory if the reference count reaches zero.

Arguments:

- freeList: a list of free memory blocks (modified by deallocateMemory).

- usedList: a list of used memory blocks (modified by deallocateMemory).

- startAddress: the starting address of the memory block to deallocate.

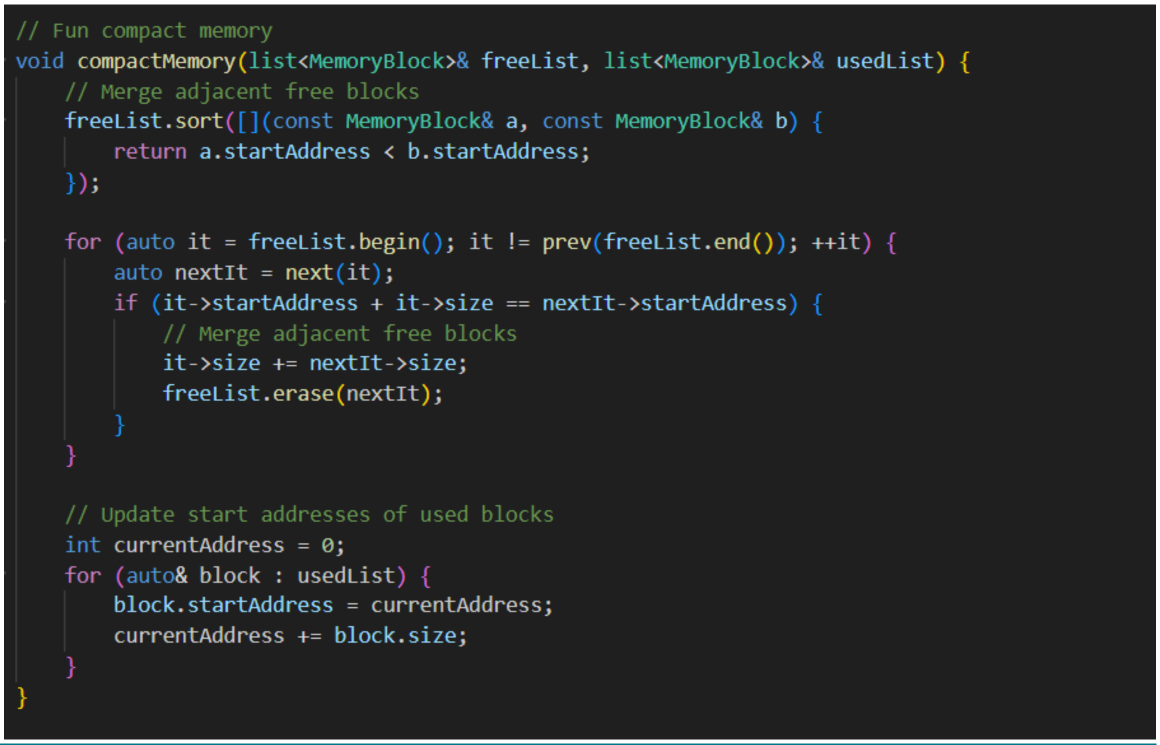
Returns:

None.

Notes:

If the reference count becomes zero, the block is added back to the free list.

- ## Function: compactMemory()



```cpp
// Fun compact memory
void compactMemory(list<MemoryBlock>& freeList, list<MemoryBlock>& usedList) {
    // Merge adjacent free blocks
    freeList.sort([](const MemoryBlock& a, const MemoryBlock& b) {
        return a.startAddress < b.startAddress;
    });

    for (auto it = freeList.begin(); it != prev(freeList.end()); ++it) {
        auto nextIt = next(it);
        if (it->startAddress + it->size == nextIt->startAddress) {
            // Merge adjacent free blocks
            it->size += nextIt->size;
            freeList.erase(nextIt);
        }
    }

    // Update start addresses of used blocks
    int currentAddress = 0;
    for (auto& block : usedList) {
        block.startAddress = currentAddress;
        currentAddress += block.size;
    }
}
```

Use:

Merges adjacent free blocks and updates start addresses of used blocks.

Arguments:

- freeList: a list of free memory blocks (modified by compactMemory).

- usedList: a list of used memory blocks (modified by compactMemory).

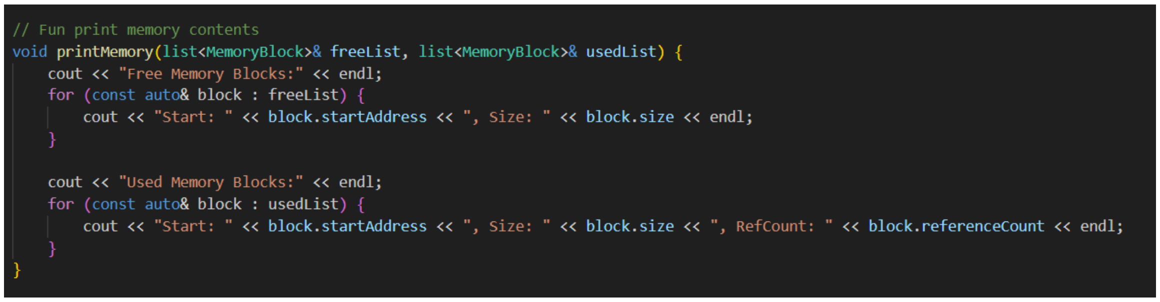
Returns:

None.

Notes:

Performs memory compaction by merging adjacent free blocks and updating start addresses.

- ## Function: printMemory()



```cpp
// Fun print memory contents
void printMemory(list<MemoryBlock>& freeList, list<MemoryBlock>& usedList) {
    cout << "Free Memory Blocks:" << endl;
    for (const auto& block : freeList) {
        cout << "Start: " << block.startAddress << ", Size: " << block.size << endl;
    }

    cout << "Used Memory Blocks:" << endl;
    for (const auto& block : usedList) {
        cout << "Start: " << block.startAddress << ", Size: " << block.size << ", RefCount: " << block.referenceCount << endl;
    }
}
```

Use:

Prints the contents of the free and used memory blocks.

Arguments:

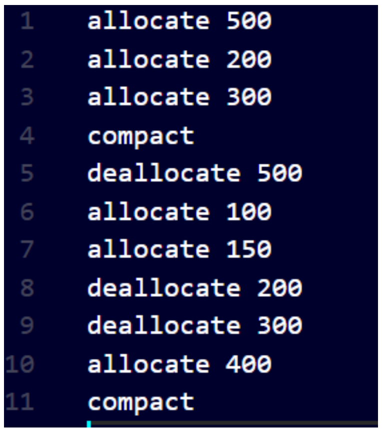
- freeList: a list of free memory blocks.

- usedList: a list of used memory blocks.

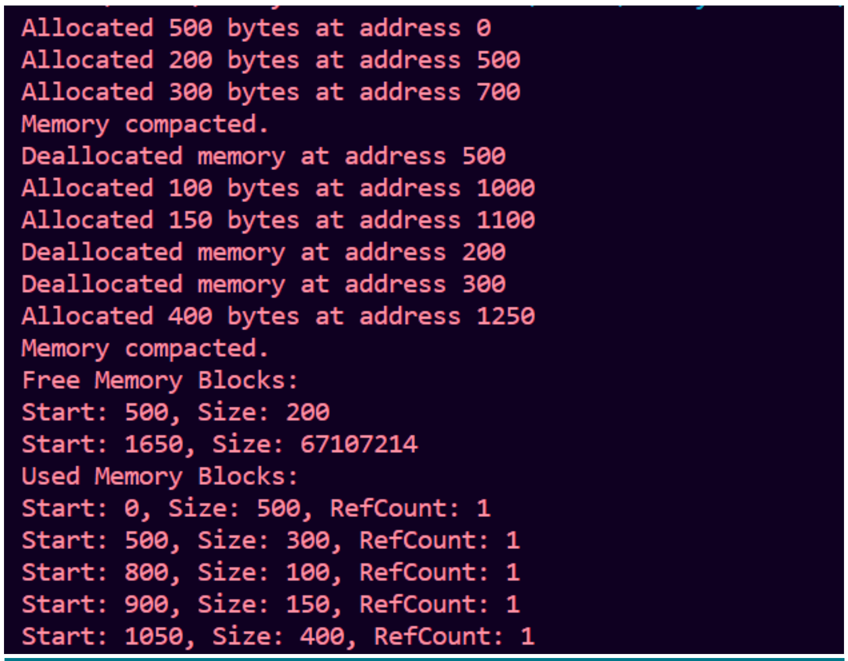
Returns:

None.

Sample Input:


```
allocate 500
allocate 200
allocate 300
compact
deallocate 500
allocate 100
allocate 150
deallocate 200
deallocate 300
allocate 400
compact
```

Output:



```
Allocated 500 bytes at address 0
Allocated 200 bytes at address 500
Allocated 300 bytes at address 700
Memory compacted.
Deallocated memory at address 500
Allocated 100 bytes at address 1000
Allocated 150 bytes at address 1100
Deallocated memory at address 200
Deallocated memory at address 300
Allocated 400 bytes at address 1250
Memory compacted.
Free Memory Blocks:
Start: 500, Size: 200
Start: 1650, Size: 67107214
Used Memory Blocks:
Start: 0, Size: 500, RefCount: 1
Start: 500, Size: 300, RefCount: 1
Start: 800, Size: 100, RefCount: 1
Start: 900, Size: 150, RefCount: 1
Start: 1050, Size: 400, RefCount: 1
```

THANK YOU!