Sabancı University

Faculty of Engineering and Natural Sciences
CS204 Advanced Programming
Fall 2024-2025

Homework 5 – Memory Management Simulation Using Templates with Binary Search Tree, Stack and Operator Overloading

Due: 30/11/2024, Saturday, 23:55

In this programming assignment, you will implement a memory management simulator that allocates and manages different variable types (**int**, **short**, and **char**) in separate memory spaces. The program uses **Binary Search Trees** to manage memory allocations and **Stack** data structures to handle variables within each allocation. You will demonstrate your understanding of using templates in C++ by implementing the solution in a templated manner. Also, some operators must be overloaded, as explained later in the document.

The main ideas that you should be familiar with before starting working on this HW are:

Classes, constructors, copy constructors, destructors, stack, binary search tree, operator overloading, template classes and template functions.

Program Structure

Your program should implement the following classes:

1. **Templated BST Class:**
    - Manages memory allocations for a specific type (int, short and char)
    - Maintains a binary search tree of memory blocks (which are stacks)
    - Each tree node contains:
        - The starting byte address for the memory region corresponding to this node.
        - Size of allocation of the memory region corresponding to this node.
        - A **Stack** for storing variables
    - Has the allocation and deallocation functionality

2. **Templated Stack Class:**
    - Manages variables within a memory block
    - Maintains LIFO (Last-In-First-Out) order
    - Handles variable definition and undefinition

Input Structure

Your program will read commands from a text file, which will be the only input to your program. The filename should be provided as a command-line argument. Each line in the input file represents a command that manipulates the memory simulation. The commands are:

1. **Allocation (+)**
    - Format: `+ startByte type size`
    - Example: `+ 1000 int 12`
    - Allocates a memory block for the specified type (the `size` must be a multiple of the size of the data type used to specialize the template).
    - Checks for overlaps with existing allocations.

2. **Variable Definition (<<)**
    - Format: `<< address type value`
    - Example: `<< 1000 int 42`
    - Defines a variable at the specified address
        - The `address` must be the starting byte for the next free memory region in the corresponding stack. Otherwise, no memory will be used.
    - Checks for:
        - Valid allocation containing the address (LIFO property of the stack).
        - No overlap with existing variables.
        - Sufficient space in allocation (if the stack has enough remaining space).
        - Maintains LIFO property.
    - **Note:** This does not update the tree nodes, but updates a stack inside a node.

3. **Variable Deletion (>>)**
    - Format: `>> address type`
        - The `address` must be the starting byte for the last element in the corresponding stack. Otherwise, no deletion operation will be performed.
    - Example: `>> 1000 int`
    - Removes the most recently defined variable in the corresponding stack.
    - Maintains LIFO property.
    - **Note:** This does not update the tree nodes, but updates a stack inside a node.

4. **Deallocation (-)**
    - Format: `- address type`
    - Example: `- 1000 int`
    - Deallocates the memory region containing the specified address (any `address` from the memory is valid).
    - **Note:** This operator deletes the tree node containing the given `address` (if it exists).

5. **Display Command**
    - Member function name: `display`
    - Shows the current state of all allocations and their variables

The Task

Given an input text file name as a command-line argument, your program should implement a templated memory management system that does the following:

- **Template Implementation Requirements:**
    - Create a **templated BST class** that can work with different data types (int, short, char)
    - Implement a **templated Stack class** for variable storage
    - Ensure proper **template syntax** and **header/implementation separation**
- **Memory Management Tasks:**
    - **Maintain separate BSTs** for each data type:

```
BST<int> intBST;    // For 4-byte integers
BST<short> shortBST;  // For 2-byte shorts
BST<char> charBST;    // For 1-byte characters
```

    - **Handle type-specific sizes:**
        - int: 4 bytes
        - short: 2 bytes
        - char: 1 byte
- **Command Processing:**
    - **Read and parse** commands from the input file using operator overloading:
- **Memory Safety Requirements:**
    - **Prevent memory overlaps**:
        - Check allocation boundaries before inserting in BST
        - Verify variable placement within stack bounds
    - **Implement LIFO property:**
        - Stack-based variable management
        - Only the most recently defined variable can be undefined
    - **Report all operations:**
        - Successful allocations/deallocations
        - Variable definitions/undefinitions
        - Error conditions
- **Template-Specific Requirements:**
    - Implement all functionality using template classes
    - Use template member functions for operations

Note about implementaions

We are giving the **main.cpp** file with the homework bundle for you to use. **Do not modify it.** You are asked to use it and implement the other needed **.cpp** and **.h** files for the program to work correctly. Please make sure to understand the **main.cpp** file before you start implementing the other needed files.

Variable Structure: Below is a struct **Variable** that you should use to represent a variable in your program

```
// Structure to hold variable information

struct Variable {

    int address;    // Memory address of the variable

    T value;       // Value of the variable

};
```

In the **main.cpp** these are overloaded operators:

- operator+= for allocation.
- operator-= for deallocation.
- operator<< for variable definition.
- operator>> for variable undefinition.
- operator[] for accessing a stack containing an address.
    - The operator[] allows access to the stack corresponding to a specific memory address in the BST. If no allocation exists at the provided address, it returns *nullptr*.

We included some comments at the beginning of the main.cpp, please check them.

Sample Runs

Some sample runs are given below, but these are not comprehensive; therefore, you have to consider **all possible cases** to get full marks. **You can assume that the commands are syntactically correct. However, the addresses inside the commands can be invalid; you do not need to report these cases. Just do nothing.**

Please do not try to understand the requirements by checking out the sample runs' details only. They may give you just an idea, but you have to read the requirements from the document to understand fully what to do. Italic file names should be given as command-line arguments in your program.

Sample Run 1 (file name: s1.txt)
File content:

+ 1000 int 12

<< 1000 int 42

<< 1004 int 84

<< 1008 int 126

display

>> 1008 int

>> 1004 int

>> 1000 int

display

- 1005 int

Expected Output

Allocated 12 bytes starting at byte 1000.

Defined variable at address 1000: Type i, Value: 42.

Defined variable at address 1004: Type i, Value: 84.

Defined variable at address 1008: Type i, Value: 126.

Current state of memory allocations:

Integer Allocations:

Memory allocation starting at byte 1000 with size 12 bytes:

Stack starting at byte 1000:

Variable from address 1008 to 1011, Type: i, Value: 126

Variable from address 1004 to 1007, Type: i, Value: 84

Variable from address 1000 to 1003, Type: i, Value: 42

Short Allocations:

Char Allocations:

Undefined variable at address 1008.

Undefined variable at address 1004.

Undefined variable at address 1000.

Current state of memory allocations:

Integer Allocations:

Memory allocation starting at byte 1000 with size 12 bytes:

Stack starting at byte 1000:

Short Allocations:

Char Allocations:

Deallocated memory containing byte 1005.

Final state of memory allocations:

Integer Allocations:

Short Allocations:

Char Allocations:

Some Important Rules

Please check and follow the homework submission rules mentioned in your lecture slides. They are included below again for your reference.

- Submission is through SUCourse only. Anything else will be ignored.
- Submit all homework-related files in a compressed archive
  - Zip format must be used. Do not use rar or other formats.
- Label your homework as
  - SUNETUserName_Lastname_Firstname_hw#.zip
  - e.g. aliko_Koduguzel_Ali_hw3.zip
  - Otherwise, we cannot find your homework
- Write your name inside the files as well
- The file types must be native (.c, .cpp. .h etc…)
  - For example, do not copy and paste your code to an MS Word file
- Any missing files in your submission will lose you points.
  - If a critical file is missing (for example, your source code) your grade may be zero.
- Do not expect too much detailed explanations in the homework specifications
  - You are now grown-ups 🙂
  - You have to think about how to design and program!

Late Submissions

- There are no late submissions unless stated otherwise.

Grading

- Homework must be compiled and run.
- Test cases will be different from the sample cases in the homework specification
- Grades will mostly be assigned based on proper functionality
  - Indentation, proper naming, comments, and efficiency issues will also be taken into consideration
- During grading, no debugging will be done by the graders to understand what is wrong with the code.
- Please read the related policy in the syllabus

Extra Rules Regarding Submission

Do NOT use any spaces or non-ASCII and Turkish characters in the file names.

The internal clock of SUCourse might be skewed by a couple of minutes, so do not leave the submission to the last minute.

It'd be a good idea to write your name and last name in the program (as a comment line, of course). Do not use any Turkish characters anywhere in your code (not even in comment parts).

To get full credit, your program must be efficient, modular (with the use of functions), well-commented, and properly indented. Besides, you also have to use understandable identifier names. The presence of any redundant computation, bad indentation, meaningless identifiers, or missing/irrelevant comments may decrease your grade in case we detect them. Moreover, we may test your programs with large test cases. Hence, take into consideration the efficiency of your algorithms other than correctness.

Please send your e-mails to "ahmed.salem@sabanciuniv.edu" for your HW-related questions.

Good Luck,

CS204 Team