Mini-Lab 1: C Basics & Address Space

COMP3230, The University of Hong Kong

Sept. 2024

Total 1 point

Objective

At the end of this lab, you will be able to:

- Review C language writing, compiling, and debugging.
- Understand the difference between variables on heap and stack.

Instructions

Before we start, you can refer to https://www.w3schools.com/c/ to review basic syntax of C language. For OS course, we will practice C language from a system perspective.

In this mini-lab, let's explore the differences between stack and heap memory allocation using C language, and investigate how to correct improper memory management.

Stack v.s. Heap

Unlike **Stack** variables, which are automatically deallocated when their containing code block (scope delineated by `{}`) terminates, **Heap** variables persist beyond the scope in which they were allocated. To prevent memory leaks, it is essential to explicitly deallocate heap variables using `free(pointer)` when they are no longer needed.

Stack is fast and efficient but limited in size and requires memory sizes to be known at compile time. It is ideal for local variables and function call management but unsuitable for large or dynamic memory needs.**heap** offers flexibility for dynamic memory allocation but is slower and requires careful manual management to avoid memory leaks and fragmentation. It is best used for large, dynamic data structures such as linked lists or trees, where the size is not known ahead of time.

How to choose? When deciding between heap and stack allocation, consider the scope and size of the data. If the data's lifetime is within a single code block, stack allocation is typically suitable. However, for data that needs to persist across multiple blocks or functions, heap allocation is more appropriate. Concerning size, smaller, fixed-size variables are usually allocated on the stack, while larger data structures or those with unpredictable sizes are better placed on the heap to avoid potential stack overflows.

In C language, heap memory can be allocated by `malloc`. The syntax is:

```
// allocate 10 int-sized memory and save the first int's location to the ↩
    pointer "numbers".
int *numbers = malloc(10 * sizeof(int));
// free the allocated memory
free(numbers);
```

Compared with stack memory allocation, simply declaring a variable within a function or block suffices. For instance:

```
int numbers[10];
```

The above sentence automatically allocates space for 10 integers on the stack. The memory is automatically released when the variable goes out of scope, contrasting with heap memory, which requires explicit deallocation using functions like `free()`.

Global Variables

In C programming, a global variable is declared outside of all functions and can be accessed and modified by any function in the program. This makes it possible for multiple functions to share the same data without needing to pass the variable as an argument.

```
#include <stdio.h>

// Global variable declaration
int globalVar;

void function1() {
    globalVar = 5;  // Assign value in function1
    printf("globalVar in function1: %d\n", globalVar);
}

void function2() {
    globalVar = 10;  // Assign value in function2
    printf("globalVar in function2: %d\n", globalVar);
}

int main() {
    printf("Initial globalVar value in main: %d\n", globalVar);

    function1();  // Call function1, modify globalVar
    function2();  // Call function2, modify globalVar

    printf("Final globalVar value in main: %d\n", globalVar);
    return 0;
}
```

Global variables are useful when multiple functions need to share or modify the same data without passing arguments. **Advantages** include easy access across functions and persistence of data throughout the program's lifetime. **Disadvantages** include the potential for unintended modifications, which can make debugging difficult. Global variables are best suited for small projects or situations where sharing state between functions is essential, but should be used sparingly to maintain code clarity and prevent side effects.

Practice!

In the provided `lab1-stack.c` file, a stack array is created and returned from a function while we will see if this behavior is legal in the compilation and execution stage.

1. Compile and run `lab1-stack.c` to observe its behavior.

   ```
   gcc lab1-stack.c -o lab1-stack
   ./lab1-stack 5  # to create an array with 5 elements
   ```

   Return a stack variable: You will notice that the program can be compiled with a warning and exits with an error when executing it.

   This is because returning a pointer to a local (stack-allocated) variable from a function is an undefined behavior in C. Although behavior is undefined, the code can be compiled. The local variable's memory is automatically deallocated when the function exits, so the returned pointer points to an invalid memory location. Accessing or modifying the memory through this pointer can lead to unpredictable results, segmentation faults, or other errors.

   Always ensure that any memory you return from a function remains valid after the function has completed its execution.

   To address this issue, we should modify the code to allocate the variable on the heap instead of the stack.

2. Complete TODO sections in `lab1-heap.c` to use heap variables instead. This will involve using `malloc` to allocate memory on the heap and `free` to deallocate it when it is no longer needed.

3. If implemented correctly, you will notice an expected output of the correct array.

Submission

(1 pt) Submit your modified code as `lab1-heap_<your_student_id>.c`.

Appendix

```c
// file: lab1-stack.c
#include <stdio.h>
#include <stdlib.h>

int* initialize_array(int n){
    int stack_arr[n];
    for (int i = 0; i < n; i++) {
        stack_arr[i] = 0;
    }

    // Note: The following return is problematic
    // because stack_arr is a local variable and will be deallocated once ↩
       the function returns.
    return stack_arr;
}
int main(int argc, char *argv[]) {
    int *arr;  // Pointer for our dynamically allocated array

    // Check if the command line argument is provided
    if (argc != 2) {
        printf("Usage: %s <number_of_elements>\n", argv[0]);
        return 1;  // Exit with an error code
    }

    // Convert the command line argument to an integer
    int n = atoi(argv[1]);

    arr = initialize_array(n);

    for(int i=0; i<n; i++){
        printf("%d\t", arr[i]);
    }
    return 0;
}
```

```c
// file: lab1-heap.c
#include <stdio.h>
#include <stdlib.h>

int* initialize_array(int n){
    int *heap_arr = NULL;
    // TODO: 1. Allocate memory for 'n' integers using malloc. Assign the ↩
       address to 'heap_arr'. (~1 line)

    if (heap_arr == NULL) {
        printf("Memory allocation failed!\n");
        return NULL;  // Exit with an error code
    }

    // TODO: 2. Use a loop to navigate through the array.
    // For each index 'i', set arr[i] to i * i (square of index). (~3 ↩
       lines)

    return heap_arr;
}

int main(int argc, char *argv[]) {
    int *arr;  // Pointer for our dynamically allocated array

    // Check if the command line argument is provided
    if (argc != 2) {
        printf("Usage: %s <number_of_elements>\n", argv[0]);
        return 1;  // Exit with an error code
    }

    // Convert the command line argument to an integer
    int n = atoi(argv[1]);

    arr = initialize_array(n);

    for(int i=0; i<n; i++){
        printf("%d\t", arr[i]);
    }

    // TODO: 3. Free the dynamically allocated memory. (~1 line)

    return 0;
}
```