08 - Recursion

Pre-Lab Exercises

Exercise 1

What is a recursive function? Explain the mechanism that handles recursive functions in computer memory.

Recursive Functions

In programming, a recursive function is one that calls itself, setting up recurring function calls until a base case terminates the loop of the function calling itself. Recursion is said to occur directly when the a function calls itself in its own body/definition, and is said to occur indirectly when the function (A) calls another function (B) in its body which in turn calls 'A'.

The mechanism for handling recursive function calls is the same as that for handling any function call in a C++ programme - **the processor stack or call stack**. The call stack is a LIFO data structure that keeps track of all the functions that have been called in a programme.

Every time a function is called in a C++ programme, it is placed at the top of the call stack. In terms of memory allocation, this means a block of memory called a 'frame stack' is created for the function. The frame stack stores the function's instructions, arguments, and, most importantly, the **calling function** and the call stack pointer - which always points to the top of the call stack - now points to this frame stack. Once the function has been executed, its frame stack is popped off the call stack and the stack pointer now points to frame stack of the caller. This is equivalent to programme control being returned to the caller.

Recursive function calls are handled by the mechanism outlined above with the only caveat being that both the caller and the called function are the same function (albeit with different arguments). Every time a recursive function calls itself, a new frame stack is created for the new instance of the recursive function, placed on top of the frame stack of the calling instance, and executed. This cycle of stacking frame stacks on top of another continues until a recursive function's base case is invoked, which stops further frame stacks from being created and starts a sequence of control being returned to previous instances of the recursive function. This eventually terminates with the frame stack of the first instance of the recursive function being 'popped off' the call stack.

Exercise 2

Differentiate between automatic type variables (the standard type, commonly used in C/C++) and static type variables.

Automatic vs Static Variables

Automatic variables are variables whose storage duration in memory is determined automatically. This means the automatic variables are created, stored in memory, and then 'deallocated' (marked for garbage collection/pointers to these variables are no longer tracked) as soon as the programme execution leaves their scope. Automatic variables can be created when the scope is reentered.

Static variables (created in C++ by prefacing the data type with the keyword '`static`') are variables whose lifetime lasts until the end of the programme (if they are global variables) or the end of the scope (if they are local variables). This means the values stored in local static variables will persist even when execution leaves their scope and the same value will be used if the scope is re-entered.

Test Programme

```
    cout << "Demonstrating automatic variables" << endl;
    for (int i = 0; i < 5; i++)
    {
        int n = 0;
        cout << ++n << " ";
    }
    cout << "\n\nDemonstrating static variables" << endl;
    for (int i = 0; i < 5; i++)
    {
        static int n = 0;
        cout << ++n << " ";
    }
```

Programme Output

```
Demonstrating automatic variables
1 1 1 1 1

Demonstrating static variables
1 2 3 4 5
END OF PROGRAMME
```

Exercise 3

Understand the working of the following program and comment on its operation (include a discussion on the differences between standard and static type variables).

```
// Static Variables and Recursion: Program demonstrates
// concept of static variable use in recursion
#include<iostream>
using namespace std;

/*a recursive function
 NOTE: This is a dangerous recursive function
 that only has inductive step - no base case*/
void recursive(void);

int main(void)
{
    recursive();

    return 0;
}

void recursive(void)
{
    int a = 10;             //reinit to 10 with every recursive call
    static int b = 10;      //value will be same as at end of last call

    cout << endl << " a = " << a << ", " << " b = " << b << endl;
    a++;
    b++;
    getchar();
    recursive();
}
```

Programme Output

```
 a = 10,  b = 10

 a = 10,  b = 11

 a = 10,  b = 12

 a = 10,  b = 13

 a = 10,  b = 14
```

Programme Explanation

This programme uses a recursive function to demonstrate the difference between static and automatic variables. The `recursive` function initializes one automatic integer variable ‘a’ and a `static` integer variable ‘b’ to the value 10. It prints the current value of the variables a and b to the console before incrementing both the variables. It then waits for the user to press any key (`getchar`) before creating a recursive call by calling itself.

As described in the comments before the function’s signature, the recursive function does not have a base case, which means it creates an infinite loop of recursive calls and has to be terminated manually.

However, this infinite recursion is made use of in differentiating `static` and automatic variables. While the automatic variable a is reinitialized to 10 with every recursive call, the value of b, the `static` variable, persists in memory and is not overwritten or reinitialized. At the end of each recursive call, b’s value is one greater than its value at the beginning of the call.

This shows that local `static` variables like b persist in memory and are only overwritten/deallocated/removed from tracking when they go out of scope. In this case, the variable b never goes out of scope because of the infinite recursion loop set up by `recursive`, which is why its value continues to increase with each recursive call.