Understanding Static vs Dynamic Memory Allocation in C++

Code Comparison: Stack vs Heap

```
#include<iostream>
using namespace std;

// Global pointer to demonstrate heap allocation
int* heapPtr;

void memoryComparison() {
    // Static allocation (stack)
    int stackVar = 10;

    // Dynamic allocation (heap)
    heapPtr = new int;  // memory stays even after function ends
    *heapPtr = 20;

    cout << "--- Inside memoryComparison() ---" << endl;
    cout << "Stack variable: " << stackVar << " (Address: " << &stackVar << ")" << endl;
    cout << "Heap variable: " << *heapPtr << " (Address: " << heapPtr << ")" << endl;
}

int main() {
    memoryComparison(); // Call the function

    cout << "\n--- After memoryComparison() ---" << endl;

    // The stack variable is gone (out of scope), but heap memory still exists
//    cout << "Stack variable: " << stackVar << " (Address: " << &stackVar << ")" << endl;
    cout << "Heap variable still accessible: " << *heapPtr << " (Address: " << heapPtr << ")" <<
endl;

    // Modify heap variable
    *heapPtr = 99;
    cout << "Modified heap variable: " << *heapPtr << endl;

    // Free heap memory
    delete heapPtr;

    return 0;
}
```

Output Explanation

```
--- Inside memoryComparison() ---
Stack variable: 10 (Address: 0x68feac)
Heap variable: 20 (Address: 0x6e1640)

--- After memoryComparison() ---
Heap variable still accessible: 20 (Address: 0x6e1640)
Modified heap variable: 99
```

Key Points:

- stackVar goes out of scope after the function — cannot be accessed.
- heapPtr still points to valid memory — must be manually deleted.

Visual Memory Diagram

Inside memoryComparison():

```
[Stack]            [Heap]
---------         --------------
stackVar: 10         0x1428e20 --> 20
heapPtr -->         (still lives on)
```

After memoryComparison() returns:

```
[Stack]            [Heap]
---------         --------------
(no stackVar)        0x1428e20 --> 20
heapPtr -->         (still valid)
```

What is a Memory Leak?

A memory leak in C++ occurs when dynamically allocated memory is not properly deallocated, leading to a gradual depletion of available memory. This happens when the programmer allocates memory using new but fails to release it using delete. Over time, these unreleased memory blocks accumulate, potentially causing performance issues or program crashes.

In C++, there is no automatic garbage collection. It means that any memory that is dynamically allocated by the programmer needs to be freed after its usage manually by the programmer. If the programmer forgets to free this memory, it will not be deallocated till the program lives and will be unavailable to other processes. This is called memory leak.

Problem:

- Causes wasted memory
- Long-running programs may crash or slow down
- Difficult to debug in large applications

মেমোরি লিক হলো এমন একটি সমস্যা, যেখানে কোনো প্রোগ্রাম হিপ (heap) থেকে মেমোরি বরাদ্দ নেয় (যেমন new দিয়ে), কিন্তু সেই মেমোরি কখনো ফিরিয়ে (free/delete) দেয় না।

এর ফলে:

- সেই মেমোরি আর ব্যবহার করা যায় না।
- কিন্তু তা এখনো র‍্যাম এ দখল করে আছে।
- অনেকক্ষণ চললে প্রোগ্রাম ধীর হয়ে যায় বা ক্র্যাশ করে।

সহজভাবে ধরুন:

ধরুন, আপনি একটি চেয়ার ভাড়া করলেন (heap থেকে memory নিলেন), কিন্তু আপনি কখনো সেই চেয়ার ফেরত দিলেন না (delete করলেন না)।
তাহলে আপনি যতবার চেয়ার ভাড়া করবেন, অফিসে চেয়ার ফুরিয়ে যাবে। এটিই মেমোরি লিক।

Code Example: Memory Leak in C++

```
#include<iostream>
using namespace std;

void createLeak() {
   int* ptr = new int(50); // memory allocated on heap
   cout << "Value inside createLeak(): " << *ptr << endl;
```

```
    // No delete here — memory leak occurs!
}

int main() {
    for (int i = 0; i < 5; ++i) {
        createLeak(); // called multiple times
    }

    cout << "Finished without freeing memory!" << endl;
    return 0;
}
```

What this code does:

- In createLeak(), a new integer is allocated on the heap with value 50.
- But the pointer ptr is local to the function.
- When the function ends, ptr is destroyed, but the heap memory remains allocated.
- This memory is inaccessible — you can't delete it anymore.
- Repeating this multiple times (in the loop) leaks memory again and again.

Why can I still access the value of a pointer after I use delete on it?

You can technically access it, but it's undefined behavior — which means the program might work, might crash, or worse, silently misbehave.

```
// how to allocate memory using the new operator example
#include <iostream>
using namespace std;

int main()
{
    // dynamically allocating an integer memory block
    int* ptr = new int;

    // storing a value at the memory pointed by ptr
    *ptr = 12;

    cout << "Value at memory pointed by ptr= " << *ptr<<endl;
    delete ptr;
    cout<<"After Deletion "<<*ptr<<endl;
    return 0;
}
```

Output [CodeBlocks 13.12 version]

```
Value at memory pointed by ptr= 12
After Deletion 10953656
```

Reference:

1. https://www.geeksforgeeks.org/memory-leak-in-c-and-how-to-avoid-it/
2. https://www.youtube.com/watch?v=F2nrej6Kjww