C’s Dynamic Memory Allocation Assisting AI Model Training Dataset AI Embedded Systems use concepts of C’s Dynamic Memory Allocation

Arav Bansal

Founder & CEO AVAUIRK (OPC) Private Limited

DOI : https://doi.org/10.51583/IJLTEMAS.2025.1412000116

Received: 01 January 2026; Accepted: 07 January 2026; Published: 12 January 2026

ABSTRACT

Embedded AI devices operate under tight resource constraints (limited RAM, CPU, and power), yet they need to run inference efficiently. Dynamic memory allocation and dynamic variables play a crucial role here. Embedded systems often use C for AI deployment, so dynamic memory management (malloc, calloc, realloc, free) is central for ‘Model Parameter Storage’, ‘Input Buffers for Sensor Data’, ‘Batch Processing and Streaming’, ‘Memory Pooling’, ‘Dynamic variables for adaptability’.

Index Terms— C, Dynamic Memory Allocation, Dynamic Variables, AI Model Training dataset, Embedded Systems

INTRODUCTION

The rapid evolution of artificial intelligence has led to unprecedented growth in model complexity and dataset sizes. Modern AI models, such as large language models (LLMs) and deep neural networks (DNNs), often require the handling of billions of parameters and terabytes of training data. This scaling places immense pressure on memory management systems, especially when training or deploying models on platforms with limited resources. C, as a low-level systems programming language, remains a preferred choice for implementing performance-critical components of AI frameworks and embedded systems due to its direct control over memory and hardware resources.

Because of its efficiency, C is frequently used in developing compiler and interpreters. The GNU Compiler Collection (GCC) and the Python's CPython interpreter are both implemented in C. Because of C's ability to provide a fast speed and portability, both of these compilers/interpreters benefit from parsing the code and executing it.

Dynamic memory allocation in C, facilitated by functions such as malloc, calloc, realloc, and free, allows programs to allocate memory at runtime, adapting to varying data sizes and computational demands. This flexibility is essential for AI workloads, where the size of input data, intermediate computations, and model parameters can change dynamically during training and inference. However, dynamic memory management introduces challenges related to performance, fragmentation, security, and determinism, particularly in embedded and real-time systems.

Embedded AI systems, including IoT devices, edge processors, and microcontrollers, operate under stringent memory and compute constraints. Efficient memory management in these environments is crucial for enabling real-time intelligence and autonomous decision-making without reliance on cloud resources. Frameworks like TensorFlow Lite Micro have pioneered memory management strategies that balance flexibility and determinism, often leveraging static allocation and custom memory planners to avoid fragmentation and ensure predictable behavior.

This paper aims to provide a comprehensive analysis of dynamic memory allocation and dynamic variables in C as they pertain to AI model training datasets and embedded AI systems.

DESCRIPTION

Large datasets in AI models often require millions of samples (images, text, sensor data). Static arrays in C (int arr[1000];) are insufficient because dataset sizes are not known at compile time. Dynamic allocation (malloc, calloc, realloc, free) allows memory to be requested at runtime, adapting to dataset size. Instead of reserving huge blocks of memory upfront, allocation can be done in chunks, reducing wasted memory. With this it provides both flexibility and improves efficiency which are fundamental needs for AI models. Dynamic memory allocation enables:

- **Scalability**: Handle datasets larger than available memory.
- **Adaptability**: Adjust to varying input sizes and model architectures.
- **Performance**: Optimize memory usage, reduce fragmentation, and support parallel training.

Embedded systems often use ‘C’ for AI deployment, so dynamic memory allocation and dynamic variables play a central role in applying:

1. **Model Parameter Storage**

- Neural network weights and biases are often too large to store statically.
- Dynamic allocation allows loading only the required parameters into memory at runtime.
- Example: Allocate memory for convolution filters only when a layer is active.

```
float *weights = (float *)malloc(num_filters * filter_size * sizeof(float));

// Use weights during inference

free(weights);
```

2. Input Buffers for Sensor Data

- Embedded AI devices (IoT cameras, wearables) receive variable-sized input streams.
- Dynamic buffers handle unpredictable data sizes (e.g., audio frames, image patches).
- This avoids wasting memory on oversized static arrays.

3. **Batch Processing & Streaming**

- Instead of loading the entire dataset, embedded devices process data in **mini-batches**.
- Dynamic allocation enables creating temporary buffers for each batch, then freeing them to conserve RAM.

4. **Dynamic Variables for Adaptability**

- Dynamic variables allow the system to adapt to changing conditions:
  - Adjusting buffer size based on available memory.
  - Scaling model complexity depending on battery level or CPU load.

- Example: A smart camera dynamically reduces image resolution if memory is low.

5. **Memory Pooling**

- Embedded AI often uses **custom memory pools** instead of raw malloc/free to reduce fragmentation.
- Pools allocate a large block once, then subdivide it for tensors, activations, and intermediate results.
- This ensures predictable performance—critical in real-time inference.

HOW DOES THIS WORK

To evaluate memory management strategies in AI workloads, we employ a combination of synthetic benchmarks and real-world applications:

Synthetic Benchmarks: Measure allocation/deallocation throughput, latency, and fragmentation under controlled conditions using tools like Mimalloc-bench, Google Benchmark, and custom C programs.

Real-world Applications: Analyze memory usage in AI model training (e.g., LLMs, CNNs) and embedded inference engines (e.g., TensorFlow Lite Micro).

Profiling Tools: Utilize Valgrind, Heaptrack, Intel VTune Profiler, and custom logging to monitor memory allocation patterns, leaks, and fragmentation.

Tools and Profiling Techniques

- **Valgrind**: Detects memory leaks, invalid accesses, and fragmentation in C programs.
- **Heaptrack**: Profiles heap memory usage, identifying hotspots and leaks.
- **RecordingMicroAllocator (TFLM)**: Audits memory usage in TensorFlow Lite Micro's tensor arena.
- **Custom Logging**: Tracks allocation and deallocation events, enabling analysis of memory usage patterns.

Design Patterns and Best Practices

- **Memory Pooling**: Pre-allocate fixed-size blocks to reduce fragmentation and allocation overhead.
- **Custom Allocators**: Implement specialized allocators (e.g., TLSF, buddy system) for predictable allocation times in real-time systems.
- **Static Allocation**: Use static buffers for critical data structures in embedded systems to ensure determinism.
- **Error Handling**: Check allocation results for NULL and handle failures gracefully.
- **Profiling and Testing**: Regularly profile memory usage and test for leaks and fragmentation.

C Memory Layout and Segmentation

C programs utilize a well-defined memory model comprising several segments:

- **Code Segment**: Stores executable instructions.
- **Data Segment**: Holds global and static variables.

- **Stack**: Manages local variables and function call data, following a Last-In, First-Out (LIFO) structure.
- **Heap**: Used for dynamic memory allocation, accessed via pointers and managed manually by the programmer.

Dynamic memory allocation occurs in the heap, enabling runtime allocation and deallocation of memory blocks. This is in contrast to static allocation, where memory is reserved at compile time and remains fixed throughout program execution.

Dynamic Memory Allocation Functions

C provides four primary functions for dynamic memory management, defined in <stdlib.h>:

Table 1: Dynamic Memory Functions

Function	**Description**	**Usage Example**
malloc	Allocates uninitialized space	int *arr = malloc(10 * sizeof(int));
calloc	Allocates zero-initialized space	int *arr = calloc(10, sizeof(int));
realloc	Resizes previously allocated space	arr = realloc(arr, 20 * sizeof(int));
free	Deallocates previously allocated space	free(arr);

These functions allow for flexible memory usage, adapting to the needs of the program at runtime.

Example: Dynamic Array Allocation

```
#include <stdio.h>

#include <stdlib.h>

int main() {

   int n, i, *ptr, sum = 0;

   printf("Enter number of elements: ");

   scanf("%d", &n);

   ptr = (int*) malloc(n * sizeof(int));

   if(ptr == NULL) {

      printf("Error! memory not allocated.");

      exit(0); }

   printf("Enter elements: ");

   for(i = 0; i < n; ++i)  {

      scanf("%d", ptr + i);

      sum += *(ptr + i); }
```

```
        printf("Sum = %d", sum);

        free(ptr);

        return 0; }
```

This example demonstrates allocating memory for an array of integers at runtime, processing user input, and freeing the memory when done.

Table 2: Static vs. Dynamic Memory Allocation

Feature	**Static Allocation**	**Dynamic Allocation**
Allocation Time	Compile-time	Runtime
Memory Area Used	Stack	Heap
Flexibility	Fixed size	Can change size at runtime
Deallocation	Automatic (by compiler)	Manual (using free)
Speed	Faster	Slightly slower
Risk of Fragmentation	Low	High
Suitability for AI	Deterministic workloads	Variable-sized datasets

Static allocation is preferred for deterministic, resource-constrained environments, while dynamic allocation offers flexibility for handling variable data sizes.

Key Benefits with use case

Scientific Computing

C supports high-performance scientific applications, including numerical simulations and supercomputing tasks. Libraries like BLAS are implemented in C for efficient matrix operations in fields like physics and climate modelling.

Real-Time Systems

C is essential for real-time systems, including aerospace control (usually flight controllers), medical devices, and industrial automation. Its deterministic performance provides precise timing within applications, such as flight controllers and robotic systems.

Role in AI Model Training: Handling Large and Variable-sized Datasets

AI model training involves processing large and often variable-sized datasets, requiring flexible memory management. Dynamic memory allocation in C enables:

Input Data Management: Allocate memory for batches of input data, which may vary in size depending on the dataset and batch configuration.

Model Parameters: Store weights, biases, and other parameters, which can scale to billions in large models.

Intermediate Computations: Allocate temporary buffers for activations, gradients, and other intermediate results during forward and backward passes.

Example: Neural Network Forward Pass in C

```
#include <stdio.h>

#include <math.h>

#include <stdlib.h>

double sigmoid(double x) { return 1.0 / (1.0 + exp(-x)); }

int main() {

    int input_size = 3;

    double *weights = (double*) malloc(input_size * sizeof(double));

    double *inputs = (double*) malloc(input_size * sizeof(double));

    double bias = 0.1, output = bias;

    // Initialize weights and inputs

    for (int i = 0; i < input_size; i++) {

      weights[i] = 0.2 + 0.3 * i;

           inputs[i] = 1.0 + i;

           output += weights[i] * inputs[i];}

     output = sigmoid(output);

     printf("Output: %.4f\n", output);

     free(weights);

     free(inputs);

     return 0;}
```

This code dynamically allocates memory for weights and inputs, demonstrating how C can manage variable-sized data structures in AI computations.

Benefits of Dynamic Allocation in AI Training

Flexibility: Adapt to varying dataset sizes and model architectures.

Efficiency: Allocate only the required memory, reducing waste.

Scalability: Support large models and datasets by allocating memory as needed.

Error Handling: Implement robust checks to prevent buffer overflows and allocation failures.

Benchmarks indicate that dynamic allocation can reduce memory overhead by up to 30% compared to fixed-size structures, enhancing performance in large-scale systems.

Performance Considerations: Speed, Latency, and Throughput

Dynamic memory allocation impacts performance in several ways:

Allocation/Deallocation Overhead: Runtime allocation incurs additional CPU cycles, potentially slowing down throughput.

Fragmentation: Repeated allocations and deallocations can fragment the heap, reducing effective memory usage and increasing allocation time.

Memory Efficiency: Fragmentation, Pooling, and Custom Allocators

Fragmentation

Fragmentation occurs when free memory is scattered in small, unusable blocks, leading to inefficient utilization. Studies show that up to 30% of usable memory can become fragmented in long-running processes.

Strategies to Counteract Fragmentation

Memory Pooling: Pre-allocate fixed-size blocks and recycle them to minimize fragmentation.

Custom Allocators: Implement allocators tailored to usage patterns (e.g., segregated lists, buddy system).

Defragmentation Algorithms: Compact memory by rearranging allocations, though this may introduce overhead and is less common in C.

Example: Memory Pool Allocation

```
#define POOL_SIZE 256

char memoryPool[POOL_SIZE];
```

Memory pools are particularly effective in embedded systems, reducing allocation overhead and fragmentation.

Case Study 1 - Embedded AI Systems: IoT, Edge, Microcontrollers

Embedded AI systems operate under severe resource constraints, necessitating efficient memory management strategies:

Static Allocation: Preferred for deterministic behavior and minimal fragmentation.

Memory Pooling: Reduces allocation overhead and fragmentation.

Custom Allocators: Implemented for real-time requirements (e.g., TLSF, Half-Fit).

Hardware Features: Utilize Memory Protection Units (MPUs) to isolate tasks and prevent unauthorized access.

Example: TensorFlow Lite Micro on Microcontrollers

TensorFlow Lite Micro runs on devices with as little as 16KB of RAM, using a pre-allocated tensor arena for all memory needs.

```
size_t tensor_arena_size = 2048;

uint8_t tensor_arena[tensor_arena_size];

tflite::MicroInterpreter interpreter(model, resolver, tensor_arena, tensor_arena_size);
```

No dynamic memory allocation occurs during inference, ensuring deterministic memory usage and avoiding fragmentation.

Real-world Deployments

Smart Sensors: Deploy lightweight CNNs for real-time anomaly detection on ARM Cortex-M MCUs with tens of KB of RAM.

IoT Devices: Use memory pooling and static allocation to manage sensor data and AI inference buffers.

Edge AI Processors: Implement custom allocators and memory planners to optimize resource usage.

TensorFlow Lite Micro and Embedded Frameworks Memory Management

TensorFlow Lite Micro (TFLM) employs a memory arena divided into head (non-persistent), temporary, and tail (persistent) sections.

Table 3: Memory Arena Structure

Section	Purpose	Allocation Strategy
Head	Shared tensor buffers (non-persistent)	GreedyMemoryPlanner
Temporary	Scoped allocations (method lifetime)	Resettable chain
Tail	Persistent allocations (model lifetime)	Recording API for auditing

TFLM uses memory planners to optimize buffer reuse, sharing memory between tensors with non-overlapping lifetimes. This approach minimizes peak memory usage and ensures efficient operation on resource-constrained devices.

Case Study 2: Real-world Examples and Deployments

Large Language Model Training

Training LLMs like GPT-3 requires managing terabytes of data and billions of parameters. Dynamic memory allocation is essential for handling variable tensor sizes, optimizer states, and intermediate computations.

STAlloc: A GPU memory allocator that reduces fragmentation by exploiting spatio-temporal regularity in allocation patterns, improving throughput by up to 32.5% and reducing memory waste by up to 43%.

Embedded Console Memory Management

The ECDC embedded console uses upfront allocation and fixed-size buffers to avoid runtime fragmentation, ensuring predictable memory usage in bare-metal systems.

Industrial Workloads

GreenMalloc optimizes allocator configurations for industrial workloads, achieving up to 4.1% reduction in average heap usage without loss of runtime efficiency.

CONCLUSION

Dynamic memory allocation and dynamic variables in C are indispensable for supporting the flexible, scalable, and efficient management of AI model training datasets and AI-enabled embedded systems. The ability to allocate memory at runtime enables handling of large and variable-sized datasets, adaptive model architectures, and efficient resource utilization in both cloud and embedded environments. However, dynamic memory management introduces challenges related to performance, fragmentation, security, and determinism, particularly in resource-constrained and real-time systems.

Embedded AI systems, such as IoT devices and microcontrollers, require specialized memory management strategies to operate within stringent resource limits. Frameworks like TensorFlow Lite Micro exemplify best practices by employing static allocation and custom memory planners to ensure deterministic behavior and avoid fragmentation. Advanced allocator implementations, including tcmalloc, jemalloc, mimalloc, and TLSF, offer optimized solutions for diverse workloads, balancing speed, scalability, and memory efficiency.

Security remains a paramount concern, with heap corruption and buffer overflows posing significant risks. Tools like CAMP, AddressSanitizer, and Valgrind provide robust mechanisms for detecting and mitigating vulnerabilities. Profiling and benchmarking tools, such as Heaptrack and Mimalloc-bench, enable developers to analyze and optimize memory usage patterns.

ACKNOWLEDGMENT

This paper is designed to reflect the importance of ‘C’, in the AI era. While it’s been thought that Python and other languages are taking over, but C still stands rock solid in terms of AI Embedded technologies, IoT and plays a pivot role in Security working at the core.

REFERENCES

Below are the reference sites which I traversed to help build my understanding:

1. Inventor of C (books, articles) by Dennis MacAlistair Ritchie & Ken Thompson (Dennis Ritchie - Wikipedia)
2. Classic Data Structures by D. Samanta - Classic Data Structures - Google Books
3. Stanford Education : Scheduling Techniques for Concurrent Systems March 24, 2000 (10)
4. Dynamic Memory Allocation and Fragmentation – by Nikola Zlatanov (link)
5. OpenAI website - OpenAI (https://openai.com)