



Data Structures and Program Design in C

Topic 8: Dynamic Memory Allocation and Linked Lists

Dr Manjusri Wickramasinghe



Outline

- Dynamic Memory Allocation
 - Memory Leaks
- Aliasing and Dangling Pointers
- register and auto variables
- Command Line Arguments
- Requirement for Dynamic Structures
- Node and Linked List Concept
 - Singly Linked List (SLL)
 - Stack and Queue Implementation using SLL

What is Dynamic Memory Allocation?

- Dynamic memory allocation is the process of assigning memory to a program at runtime, giving the programmer the control and capability to work with data structures of varying sizes.
 - Increases flexibility
 - Increases efficiency
- Dynamic memory allocation occurs both in the stack and the heap memory.
- When memory is allocated for non-function call-based aspects, the memory is assigned from the heap.

Dynamic Memory Allocation in C

- The C programming language provides four (4) functions to handle dynamic memory allocation.
- These functions are defined in the standard library <stdlib.h> header and the four functions are as follows:
 - malloc (...)
 - calloc (...)
 - free (...)
 - realloc (...)

size_t data type ...(1)

- Is a type defined in stddef.h and used in headers such as stdio.h, stdlib.h, time.h and string.h.
- size_t guarantees to be big enough to handle the largest possible object in the host system.
 - In a 32-bit system, it is equivalent to an unsigned int
 - In a 64-bit system, it is equivalent to an unsinged long long.
- size_t is never negative.

`size_t` data type ...(2)

- Use of the `size_t` data type ensures the portability of the code.
- Use of `size_t` states to the reader of a program that it deals with sizes and not any other type of integers.
- Enables the easy integration with many libraries and API as it is widely used.

malloc (...) ...(1)

- The malloc stands for memory allocation.
- The functions reserve a block of memory of a specified size from the heap and return a pointer.

```
void *malloc(size_t size);
```

- The call returns a void pointer that can be cast into a pointer of any type.

malloc (...) ...(2)

- The function does not initialize memory at execution time, and as such, garbage values present at the time of allocation are the initial values of the memory block.
- If no sufficient space is available to allocate, a null pointer is returned.
- Size parameter expects the size of the memory block required in bytes.

`calloc(...)` ...(1)

- The `calloc` stands for “contiguous allocation.”
- Performs similar to `malloc(...)` with the following key differences
 - Initializes each block of memory so allocated to zero.
 - It requires two parameters: the total number of elements (`n`) and the size of elements.
 - Returns a void pointer.

```
void *calloc(size_t nelem, size_t elsize);
```

calloc (...) ...(2)

- calloc (...) is a function that assigns a specified number of blocks of memory to a single pointer, while malloc (...) creates a single block of memory.
- malloc (...) is faster than calloc (...) .
- calloc (...) initializes the memory to zero while malloc (...) does not.

free (...)

- The `free (...)` method is used to deallocate memory that was allocated by `malloc (...)` and `calloc (...)`.

```
void free(void *ptr)
```

- It is a recommended practice to free memory that has been allocated.

realloc(...)

- realloc (...) enables the reallocation of previously allocated memory when the memory allocated by malloc (...) and calloc (...) is insufficient.

```
void *realloc(void *ptr, size_t size);
```

- Returns a null pointer if the required space cannot be allocated.

Memory Leaks ...(1)

- Memory leaks occur when the programmer allocates memory in the heap and forgets to delete it, leaving such memory allocated as long as the program runs.
 - For programs that never terminate, this can be a major problem.
- Memory leaks reduce the usable heap memory of a computer and may result in a crash or system performance issues if too much memory becomes allocated.

Memory Leaks ... (2)

- Memory leaks, such as those detected during code reviews, where each allocation and deallocation is reviewed, can be detected manually, a process that involves significant effort and time.
- Automated tools enable much easier detection of memory leaks in large code bases. (e.g. Valgrind)
- Valgrind provides tools to profile and debug programs
 - To install
`$sudo apt install valgrind`
 - To detect leaks
`$valgrind --leak-check=full <executable>`

Aliasing

- Aliasing occurs when the same memory location is accessed using different names.

...

```
int *a = malloc(sizeof(int));
int *b = a;
*a = 25;
*b = 45;
printf("%d\n", *a); //??
```

...

Dangling Pointers

- When a pointer points to a deleted or freed memory location, it is called a dangling pointer.
- Dangling pointers can lead to unpredictable program behaviour.
- Can occur as a result of
 - Memory deallocation
 - Scoping issues

register Keyword ...(1)

- Registers are memory units that are built directly into the CPU. These are very fast compared to memory access.
- It is a practice to put variables frequently used in registers for faster access.
- Though the register keyword is used with a variable, the compiler ultimately decides whether to include the variable in a register.

register Keyword ...(2)

- Usage

```
register <data_type> <var_name> = <value>;
```

- Facts to note

- Accessing the address of a register (use of &) may result in an error depending on the compiler.
- Register is a storage that cannot be static or global.
- Register can only be used in local context.

auto Keyword ...(1)

- The auto keyword specifies a storage duration for a given local variable; the memory is allocated from the stack.
- Since, by default, all memory for functions is allocated using the stack, the auto keyword has no practical usage.
- Usage

```
auto <data_type> <variable_name>;
```

auto Keyword ...(2)

- The auto in C and C++ have entirely different meanings and should not be confused.
- In C, auto means a storage class without a practical use.
- In classic C++, before the C++ 11 standard, auto had the same meaning as C.
- In modern C++, which is C++ 11 and later, auto means the compiler has to infer the variables type from the call-chain using the type of value used as the initializer.

Storage Classes

- In C programming language, there are four (4) storage classes:
 - auto
 - Stored in the stack, the variables are valid within the local scope.
 - static
 - Stored in the data segment and the variables are valid until the end of the program.
 - extern
 - Stored in the data segment and the variables are valid till the program's end and across multiple files.
 - register
 - Stored in a CPU register and the variables are valid until the end of the defined block.

Command Line Arguments

- Command-line arguments are handled by the `main(...)` functions of the C program.
- To pass command-line arguments, two parameters are defined in the `main` function. These are the number of command line arguments (`argc`) and the list of command line arguments (`argv`).

```
int main(int argc, char* argv[])
{
    ...
}
```

Requirements for Dynamic Structures ...(1)

- When considering arrays, it was found that the array size has to be known in advance to allocate the relevant amount of memory.
- Such static allocation results in wastage of memory in some cases and inefficiency when the array size has to be increased to support a large set of array elements.
- To address the above shortcomings, a dynamic structure that allocates memory when required and can dynamically resize is required.

Requirements for Dynamic Structures ...(2)

- To accommodate dynamic resizability, a basic block is required with a data component and links to the other elements such that all blocks can be linked to each other, forming a data structure.
- The basic block, **a node**, contains a *data component* and *pointers to the other nodes*.
- By linking nodes to other nodes, **various data structures can be created** based on how the pointers behave.

Node ...(1)

- Node is a basic block allocated from heap storage (via malloc) where and when required to store data.
- Nodes are usually done using structs.

```
struct Node{  
    [<data_type> <member_name>]+;  
    [struct Node* <pointer_name>]+;  
}
```

Node ... (2)

- Nodes can be dynamically created by using the malloc (...) function as below.

```
void *temp = malloc(sizeof(struct Node));
```

- To add data or to initialize pointers in the newly created Node, a pointer to structure notation is used.

Linked Lists ...(1)

- Based on how nodes are chained, various data structures can be developed.
- When nodes are chained sequentially, a linked list is produced.
- The behaviour of the pointers and the number of pointers a struct Node provides three main variants of the linked list as:
 - Singly Linked List
 - Doubly Linked List
 - Circular Linked List – Single and Double variants possible

Linked Lists ...(2)

- Nodes are created in the heap memory, and a void pointer is provided to access each.
- However, when a sequential linked structure is created it is important to keep track of the beginning of the linked structure and the end of the linked structure as it does not have natural first and last elements as in the case of an array.
- As such two pointers named **head** and **tail** are kept to the first elements and the last element of the linked list respectively.

Singly Linked List ...(1)

- Singly linked list is data structure where there is a single link between elements which enables a give node to point the next node in the list.
- There are two (2) variants of inserts and delete for a linked list. These are:
 - AddToHead
 - AddToTail
 - DeleteFromHead
 - DeleteFromTail

Singly Linked List ...(2)

- **AddToHead(X)**
 - Create a new node (Temp) and insert the value X.
 - Set Temps' next reference to the current head node.
 - Update the Head reference to Temp.
- **AddToTail(X)**
 - Create a new node (temp) and insert the value X.
 - Set Tail nodes next reference to Temp.
 - Update the Tail reference to the Temp node.
- What are the special cases for insertion?

Singly Linked List ...(3)

- **DeleteFromHead()**
 - Record the data value of the Head in a temporary variable.
 - Update Head reference to the next element of the Head.
 - Return the data.
- **DeleteFromTail()**
 - Record the data value of the Tail in a temporary variable.
 - Find the node before the Tail.
 - Update the Tail reference to the node before the Tail.
- What are the special cases for deletion?

Singly Linked List ...(4)

- **Insert(X, Location, Pre/Post) //Generic Insert**
 - Find the node at the location
 - Add the new Temp node with value X as the next of the node at the location and Temp's next the location next node in the case of 'Post'
 - Add the new Temp node as next to the previous node of the location and Temp's next to the location Node in the case of 'Pre'

Singly Linked List ...(5)

- **Delete(X)**
 - Search for the node with data value X.
 - Set the Node's previous's next element to the Node's next element.
 - X can also be the node's location in the linked list.
- Other operations of Singly Linked Lists
 - Search(X)
 - PrintList()

Singly Linked List ...(6)

- Implementation of a Stack using Singly Linked List
 - `push (x)` → AddToHead or AddToTail
 - `pop ()` → DeleteFromHead or DeleteFromTail
- Implementation of a Queue using Singly Linked List
 - `enqueue (X)` → AddToHead or AddToTail
 - `dequeue ()` → DeleteFromTail or DeleteFromHead

Doubly Linked List

- The doubly linked list maintains a link for the next node and the previous node through the **next** and **prev** pointers.
- Node structure

```
struct Node{  
    [<data_type> <member_name>]+;  
    struct Node* next;  
    struct Node* prev;  
}
```

- How would the insert and delete operations change when the pointer to the previous nodes are present?

Circular Linked List

- The circular linked list can be both a singly linked list or a doubly linked list.
- The main difference between the singly linked list, doubly linked list, and the circular linked list is that the tail is connected to the head and vice versa.

Questions?