MEMORY MANAGEMENT

Memory is an important resource that must be carefully managed. While the average home computer nowadays has two thousand times as much memory as the IBM 7094 (the largest computer in the world in the early 1960s), programs and the data they are expected to handle have also grown tremendously. To paraphrase Parkinson's law, ''Programs and their data expand to fill the memory available to hold them.''

Memory management is a critical aspect of systems programming, ensuring that programs use memory efficiently and safely. It involves handling ***memory allocation*** and ***deallocation*** to ensure efficient use of available memory and avoid problems such as ***memory leaks*** and ***fragmentation***. It includes both manual and automatic techniques for managing memory in programming.

1. Allocating memory

There are two ways that memory gets allocated for data storage:

1. Compile Time (or static) Allocation
   - Memory for named variables is allocated by the compiler
   - Exact size and type of storage must be known at compile time
   - For standard array declarations, this is why the size has to be constant
2. Dynamic Memory Allocation
   - Memory allocated "on the fly" during run time
   - dynamically allocated space usually placed in a program segment known as the *heap* or the *free store*
   - Exact amount of space or number of items does not have to be known by the compiler in advance.
   - For dynamic memory allocation, pointers are crucial

Dynamic Memory Allocation

- We can dynamically allocate storage space while the program is running, but we cannot create new variable names "on the fly"

- For this reason, dynamic allocation requires two steps:
    1. Creating the dynamic space.
    2. Storing its **address** in a **pointer** (so that the space can be accessed)
- To dynamically allocate memory in C++, we use the **new** operator.
- De-allocation:
    - Deallocation is the "clean-up" of space being used for variables or other data storage
    - Compile time variables are automatically deallocated based on their known extent (this is the same as scope for "automatic" variables)
    - It is the programmer's job to deallocate dynamically created space
    - To de-allocate dynamic memory, we use the **delete** operator

Allocating space with new

- To allocate space dynamically, use the unary operator **new**, followed by the *type* being allocated.
- new int;   // dynamically allocates an int
- new double;   // dynamically allocates a double
- If creating an array dynamically, use the same form, but put brackets with a size after the type:
- new int[40];   // dynamically allocates an array of 40 ints
- new double[size]; // dynamically allocates an array of size doubles, note that the size can be a variable
- These statements above are not very useful by themselves, because the allocated spaces have no names! BUT, the new operator returns the starting address of the allocated space, and this address can be stored in a pointer:
- int * p;   // declare a pointer p
- p = new int;   // dynamically allocate an int and load address into p
-
- double * d;   // declare a pointer d

- d = new double; // dynamically allocate a double and load address into d
-
- // we can also do these in single line statements
- int x = 40;
- int * list = new int[x];
- float * numbers = new float[x+10];

Notice that this is one more way of *initializing* a pointer to a valid target (and the most important one).

Accessing dynamically created space

- So once the space has been dynamically allocated, how do we use it?
- For single items, we go through the pointer. Dereference the pointer to reach the dynamically created target:
- int * p = new int;   // dynamic integer, pointed to by p
-
- *p = 10;   // assigns 10 to the dynamic integer
- cout << *p;   // prints 10
- For dynamically created arrays, you can use either pointer-offset notation, or treat the pointer as the array name and use the standard bracket notation:
- double * numList = new double[size];   // dynamic array
-
- for (int i = 0; i < size; i++)
-     numList[i] = 0;   // initialize array elements to 0
-
- numList[5] = 20;   // bracket notation
- *(numList + 7) = 15;   // pointer-offset notation
-   //   means same as numList[7]

Deallocation of dynamic memory

- To deallocate memory that was created with new, we use the unary operator **delete**. The one operand should be a pointer that stores the address of the space to be deallocated:
- int * ptr = new int;   // dynamically created int
- // ...
- delete ptr;   // deletes the space that ptr points to

Note that the pointer ptr *still exists* in this example. That's a named variable subject to scope and extent determined at compile time. It can be reused:

```
 ptr = new int[10];          // point p to a brand new array
```

- To deallocate a dynamic array, use this form:
- delete [] *name_of_pointer*;

Example:

```
 int * list = new int[40];    // dynamic array


 delete [] list;          // deallocates the array
 list = 0;                    // reset list to null pointer
```

After deallocating space, it's always a good idea to reset the pointer to null unless you are pointing it at another valid target right away.

- **To consider:** So what happens if you fail to deallocate dynamic memory when you are finished with it? (i.e. why is deallocation important?)

Functions for Dynamic Memory Allocation in C

1. **malloc**:
    - Allocates a specified number of bytes.
    - Returns a pointer to the allocated memory or NULL if the allocation fails.

void *malloc(size_t size);

2. **calloc**:

    - Allocates memory for an array of elements, initializing all bytes to zero.
    - Returns a pointer to the allocated memory or NULL if the allocation fails.

```
void *calloc(size_t num, size_t size);
```

3. **realloc**:
    - Resizes a previously allocated memory block.
    - Returns a pointer to the newly allocated memory, which may be the same as the original or a new location.

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

4. **free**:
    - Frees previously allocated memory, making it available for future allocations.
    - Does not return a value.

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

Example Program for Dynamic Memory Allocation in C:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  int *arr;
  int n = 5;

  // Allocate memory for n integers
  arr = (int *)malloc(n * sizeof(int));
  if (arr == NULL) {
    printf("Memory allocation failed\n");
    return 1;
  }
```

```
    // Initialize and print the array
    for (int i = 0; i < n; i++) {
        arr[i] = i + 1;
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Reallocate memory for n*2 integers
    arr = (int *)realloc(arr, n * 2 * sizeof(int));
    if (arr == NULL) {
        printf("Memory reallocation failed\n");
        return 1;
    }

    // Initialize and print the new part of the array
    for (int i = n; i < n * 2; i++) {
        arr[i] = i + 1;
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Free the allocated memory
    free(arr);

    return 0;
}
```

Example Program for Dynamic Memory Allocation in Assembly

In assembly, dynamic memory allocation can be managed using system calls. For Linux, the brk and mmap system calls are often used.

```
section .data
    size    dd 1024        ; Allocate 1024 bytes

section .bss
    mem     resb 1         ; Just a placeholder for memory allocation

section .text
    global _start

_start:
    ; Allocate memory using brk
    mov eax, 45            ; sys_brk system call number
    mov ebx, mem
    add ebx, [size]
    int 0x80               ; Perform system call

    ; Check if allocation succeeded
    cmp eax, -1
    je allocation_failed

    ; Now eax contains the new program break address (end of allocated memory)
    ; Perform some operations with allocated memory (just an example)
    mov byte [mem], 0x55   ; Set the first byte to 0x55

    ; Free the allocated memory by resetting the program break
    mov eax, 45
    mov ebx, mem
    int 0x80

    ; Exit the program
    mov eax, 1
    xor ebx, ebx
    int 0x80

allocation_failed:
    ; Handle allocation failure
    mov eax, 1
    mov ebx, 1          ; Exit with error code 1
    int 0x80
```

Application Example: Dynamically resizing an array

If you have an existing array, and you want to make it bigger (add array cells to it), you cannot simply append new cells to the old ones. Remember that arrays are stored in consecutive memory, and you never know whether or not the memory immediately after the array is already allocated for something else. For that reason, the process takes a few more steps. Here is an example using an integer array. Let's say this is the original array:

```
int * list = new int[size];
```

I want to resize this so that the array called **list** has space for 5 more numbers (presumably because the old one is full).

There are four main steps.

1. Create an entirely new array of the appropriate type and of the new size. (You'll need another pointer for this).

   ```
   int * temp = new int[size + 5];
   ```

2. Copy the data from the old array into the new array (keeping them in the same positions). This is easy with a for-loop.
   ```
    for (int i = 0; i < size; i++)
       temp[i] = list[i];
   ```
3. Delete the old array -- you don't need it anymore! (Do as your Mom says, and take out the garbage!)
   ```
   delete [] list;  // this deletes the array pointed to by "list"
   ```
4. Change the pointer. You still want the array to be called "list" (its original name), so change the list pointer to the new address.
   ```
   list = temp;
   ```

That's it! The list array is now 5 larger than the previous one, and it has the same data in it that the original one had. But, now it has room for 5 more items.

2. Memory Leaks

Memory leaks occur when a program allocates memory but fails to deallocate it after use. This leads to wasted memory resources and can eventually cause the program to run out of memory.

Example of a Memory Leak in C

```
#include <stdio.h>
#include <stdlib.h>

void create_memory_leak() {
   int *ptr = (int *)malloc(sizeof(int) * 100);
   // Memory allocated but never freed
}

int main() {
   create_memory_leak();
   return 0;
```

}

Example of a Memory Leak in Assembly

In assembly, a memory leak occurs if the allocated memory is not properly deallocated. Below is an example illustrating this concept.

```
section .data
    size    dd 1024        ; Allocate 1024 bytes

section .bss
    mem     resb 1         ; Just a placeholder for memory allocation

section .text
    global _start

_start:
    ; Allocate memory using brk
    mov eax, 45            ; sys_brk system call number
    mov ebx, mem
    add ebx, [size]
    int 0x80               ; Perform system call

    ; Simulate work without freeing the memory
    mov byte [mem], 0x55   ; Set the first byte to 0x55

    ; Exit without freeing the allocated memory
    mov eax, 1
    xor ebx, ebx
    int 0x80
```

3. Garbage Collection

Garbage collection is an automatic memory management technique that identifies and frees unused memory, eliminating the need for manual deallocation.

a. Types of Garbage Collection

1. **Reference Counting**:
   - Keeps a count of references to each memory block.
   - Frees the block when the count reaches zero.
   - Limitation: Cannot handle cyclic references.
2. **Mark-and-Sweep**:
   - Marks all reachable objects starting from root references.
   - Sweeps through memory to collect unmarked objects.
   - Suitable for managing cyclic references.
3. **Generational Garbage Collection**:
   - Divides objects into generations based on their age.
   - Focuses on collecting younger objects more frequently as they are more likely to become garbage.

Example of Reference Counting in C++

```
#include <iostream>
#include <memory>

class MyClass {
public:
  MyClass() { std::cout << "Constructor called\n"; }
  ~MyClass() { std::cout << "Destructor called\n"; }
};

int main() {
  std::shared_ptr<MyClass> ptr1 = std::make_shared<MyClass>();
```

```
    std::shared_ptr<MyClass> ptr2 = ptr1; // Reference count increased
    {
        std::shared_ptr<MyClass> ptr3 = ptr2; // Reference count increased
    } // ptr3 goes out of scope, reference count decreased
    // ptr1 and ptr2 go out of scope, reference count reaches zero, memory freed

    return 0;
}
```

Example of Reference Counting in Assembly

Implementing garbage collection directly in assembly is complex, but we can simulate reference counting example above.

```
section .data
    ref_count  dd 1        ; Reference count
    size       dd 4        ; Allocate 4 bytes (int size)

section .bss
    mem        resb 4      ; Placeholder for allocated memory

section .text
    global _start

_start:
    ; Allocate memory using brk
    mov eax, 45            ; sys_brk system call number
    mov ebx, mem
    add ebx, [size]
    int 0x80               ; Perform system call

    ; Initialize allocated memory and increase ref count
    mov dword [mem], 1234  ; Set the value to 1234
    mov eax, [ref_count]
    inc eax
    mov [ref_count], eax

    ; Simulate removing a reference
    mov eax, [ref_count]
    dec eax
    mov [ref_count], eax

    ; Check ref count and free memory if 0
    cmp dword [ref_count], 0
    jne skip_free

    ; Free memory by resetting program break
    mov eax, 45
    mov ebx, mem
    int 0x80

skip_free:
    ; Exit the program
    mov eax, 1
    xor ebx, ebx
    int 0x80
```

4. Memory Fragmentation

Memory fragmentation occurs when free memory is divided into small, non-contiguous blocks, making it difficult to allocate large contiguous memory blocks.

a. Types of Fragmentation

1. **External Fragmentation**:
    - Free memory blocks are scattered outside allocated regions.
    - Reduces the availability of contiguous memory.
2. **Internal Fragmentation**:
    - Allocated memory blocks have unused space within them.
    - Occurs when memory allocation granularity is larger than required.

b. Solutions to Fragmentation

1. **Compaction**:
    - Moves allocated memory blocks together, reducing external fragmentation.
    - Can be resource-intensive.
2. **Segmentation**:
    - Divides memory into segments based on logical divisions.
    - Helps manage memory more efficiently but requires support from the OS.
3. **Paging**:
    - Divides memory into fixed-size pages.
    - Eliminates external fragmentation but may introduce internal fragmentation.

Simulating Memory Fragmentation in C

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  // Simulating fragmentation by allocating and freeing different-sized blocks
  char *block1 = (char *)malloc(100);
  char *block2 = (char *)malloc(200);
  char *block3 = (char *)malloc(50);
```

```
    free(block2); // Freeing middle block creates a gap

    char *block4 = (char *)malloc(150); // May not fit in the gap, causing fragmentation

    // Free remaining blocks
    free(block1);
    free(block3);
    free(block4);

    return 0;
}
```

Simulating Memory Fragmentation in Assembly

Memory fragmentation can be simulated by allocating and freeing blocks of different sizes

```
section .data
    size1  dd 1024        ; First block size
    size2  dd 2048        ; Second block size
    size3  dd 512         ; Third block size

section .bss
    mem1   resb 1         ; Placeholder for first block
    mem2   resb 1         ; Placeholder for second block
    mem3   resb 1         ; Placeholder for third block

section .text
    global _start

_start:
```

```
; Allocate first block
mov eax, 45          ; sys_brk system call number
mov ebx, mem1
add ebx, [size1]
int 0x80

; Allocate second block
mov eax, 45
mov ebx, mem2
add ebx, [size2]
int 0x80

; Free the second block
; Note: Normally, we'd use a proper memory manager to handle free blocks

; Allocate third block which may cause fragmentation
mov eax, 45
mov ebx, mem3
add ebx, [size3]
int 0x80

; Exit the program
mov eax, 1
xor ebx, ebx
int 0x80
```