Chapter 3: Memory Management

3.1 Memory Basics

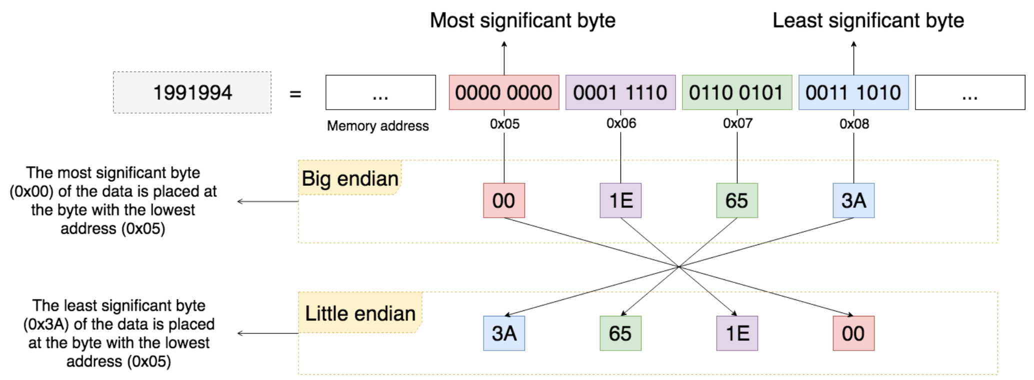
<u>Little-Endian Systems</u>: the least significant byte (LSB) is placed at the lowest memory address.

- When a multi-byte data (eg. a 32-bit integer) is stored in memory the LSB is stored at the lowest address
- Moving to higher memory address, you encounter progressively more significant bytes
    - Let's say we have the hexadecimal number 0x12345678 stored at the memory address 0x00:
        - 0x00: 0x78 (LSB)
        - 0x01: 0x56
        - 0x02: 0x34
        - 0x03: 0x12 (MSB)

<u>Big-Endian Systems</u>: the most significant byte (MSB) is placed at the lowest memory address.

- When a multi-byte data is stored in memory, the MSB is stored at the lowest address.
- Moving to higher memory address, you encounter progressively more significant bytes
    - Let's say we have the hexadecimal number 0x12345678 stored at the memory address 0x00:
        - 0x00: 0x12 (MSB)
        - 0x01: 0x34
        - 0x02: 0x56
        - 0x03: 0x78 (LSB)

Example:


Most significant byte
Least significant byte
1991994
=
...
0000 0000
0001 1110
0110 0101
0011 1010
...
Memory address
0x05
0x06
0x07
0x08
The most significant byte (0x00) of the data is placed at the byte with the lowest address (0x05)
Big endian
00
1E
65
3A
The least significant byte (0x3A) of the data is placed at the byte with the lowest address (0x05)
Little endian
3A
65
1E
00

3.2 The Stack

The “stack” in CS is a data structure used to manage function calls and local variables

- When a function call is made, a stack frame is created on the stack
- It includes:
  - The address of the return instruction
  - Parameters passed to the function
  - Space for local variables used within the function
- As functions executes, the local variables and other data are added to the stack frame
- When the function returns, the stack frame is “popped off” the stack, effectively deallocating the space it occupied

LIFO order:

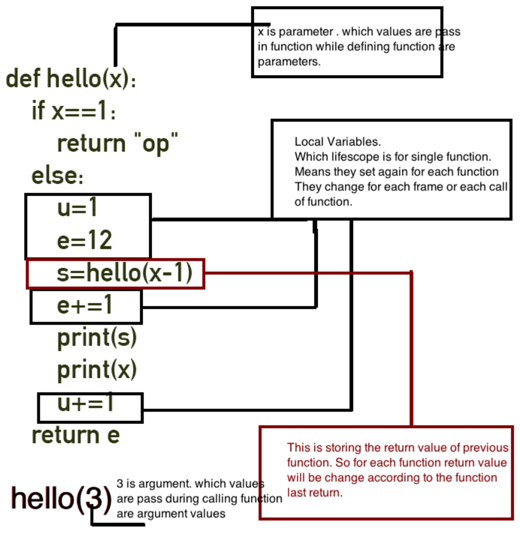
- This means that the most recently created stack frame is the first one to be removed when a function returns
- LIFO order ensures that functions are called and returned in the reverse order in which they were called, allowing for nested function calls

Stack Pointer

- Points to the current stack frame
- Keeps track of the top of the stack, allowing the program to push new stack frames onto the stack and pop them off when functions return

Deallocated Memory

- It's important to note that when memory is deallocated (e.g., when a stack frame is popped), it is not necessarily cleared or erased.
  - Deallocated memory simply becomes available for reuse, and its contents may be overwritten by new data later on.
  - This is an important consideration because accessing data in deallocated memory can lead to undefined behavior or security vulnerabilities.



```
def hello(x):
  if x==1:
    return "op"
  else:
    u=1
    e=12
    s=hello(x-1)
    e+=1
    print(s)
    print(x)
    u+=1
  return e

hello(3)
```

x is parameter . which values are pass in function while defining function are parameters.

Local Variables.
Which lifescope is for single function.
Means they set again for each function
They change for each frame or each call of function.

This is storing the return value of previous function. So for each function return value will be change according to the function last return.

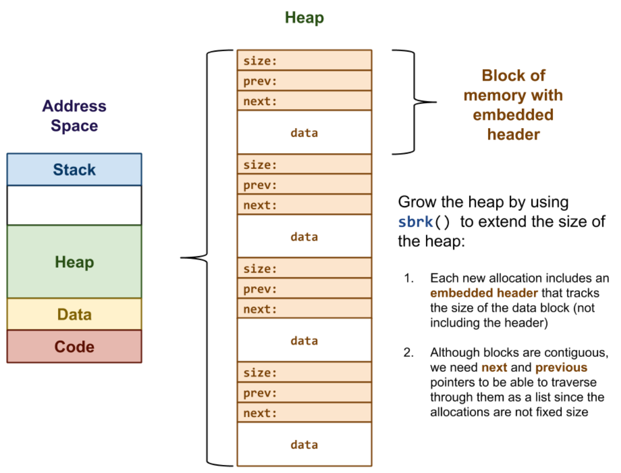
3 is argument. which values are pass during calling function are argument values

3.3 The Heap

The heap is a region of memory separate from the stack

Heap Fragmentation

- Occurs when most of the free memory in the heap is divided into many small, non-contiguous chunks
- This is problematic because, even if the cumulative free space on the heap is sufficient for a large object, if there are no open, contiguous spaces available, it becomes impossible to allocate the object.



Grow the heap by using `sbrk()` to extend the size of the heap:

1. Each new allocation includes an **embedded header** that tracks the size of the data block (not including the header)
2. Although blocks are contiguous, we need **next** and **previous** pointers to be able to traverse through them as a list since the allocations are not fixed size

Implementation of Heap:

- Every block in the heap has a header containing its size and a pointer to the next block.
- Free blocks of memory in the heap are organized as a circular linked list.
- When memory needs to be allocated in the heap, this linked list is searched to find an appropriate block.
- When memory is freed, adjacent empty blocks are combined (coalesced) into a single larger block.

Allocation and Freeing Strategies:

- Best-Fit Allocation:
  - The entire linked list of free blocks is searched to find the smallest block that is large enough to accommodate the memory requirement. This strategy minimizes wasted memory but may require searching through the entire list.
- First-Fit Allocation:
  - The first block encountered in the free list that is large enough to satisfy the memory requirement is returned.This strategy is faster but can lead to more fragmentation.
- Next-Fit Allocation:

    - Next-fit allocation is similar to first-fit, but the memory manager remembers where it left off in the free list and resumes searching from there. It strikes a balance between best-fit and first-fit strategies.

3.4 Heap Management

Using *malloc()*

- As a C programmer, you are responsible for managing heap memory.
- This is achieved through the malloc function.

Malloc Function

- Syntax: *void* malloc(size_t size)*
- *malloc* takes a size in bytes as its argument.
- It returns a pointer to the allocated memory in the heap.
- If there is no available space in the heap, it returns *NULL*.

Memory Management Guidelines:

Things to Watch Out For:

- <u>Dangling Reference</u>: Avoid using a pointer before allocating memory with malloc. This can lead to undefined behavior.
- <u>Memory Leak</u>: Always remember to free dynamically allocated memory using the free function when it's no longer needed. Failing to do so can result in memory leaks.
- <u>Avoid Freeing the Same Memory Twice</u>: Ensure that you do not call free on the same memory location more than once.
- <u>Don't Use *free* on Something Not Created with *malloc*</u>: Only use free on memory that was allocated using malloc.
- <u>*Malloc* Does Not Overwrite Existing Data</u>: Keep in mind that malloc does not overwrite or clear the content of the current memory location; it allocates new memory.

Using the *sizeof* Function:

- To determine the size of memory to allocate with malloc, it's common to use the sizeof function
- Syntax: *sizeof(type)* → *sizeof* returns the size in bytes of the given type or object.
- This allows you to write code that is architecture-agnostic (e.g., code that works on both 32-bit and 64-bit systems) by dynamically determining the size needed for an object.