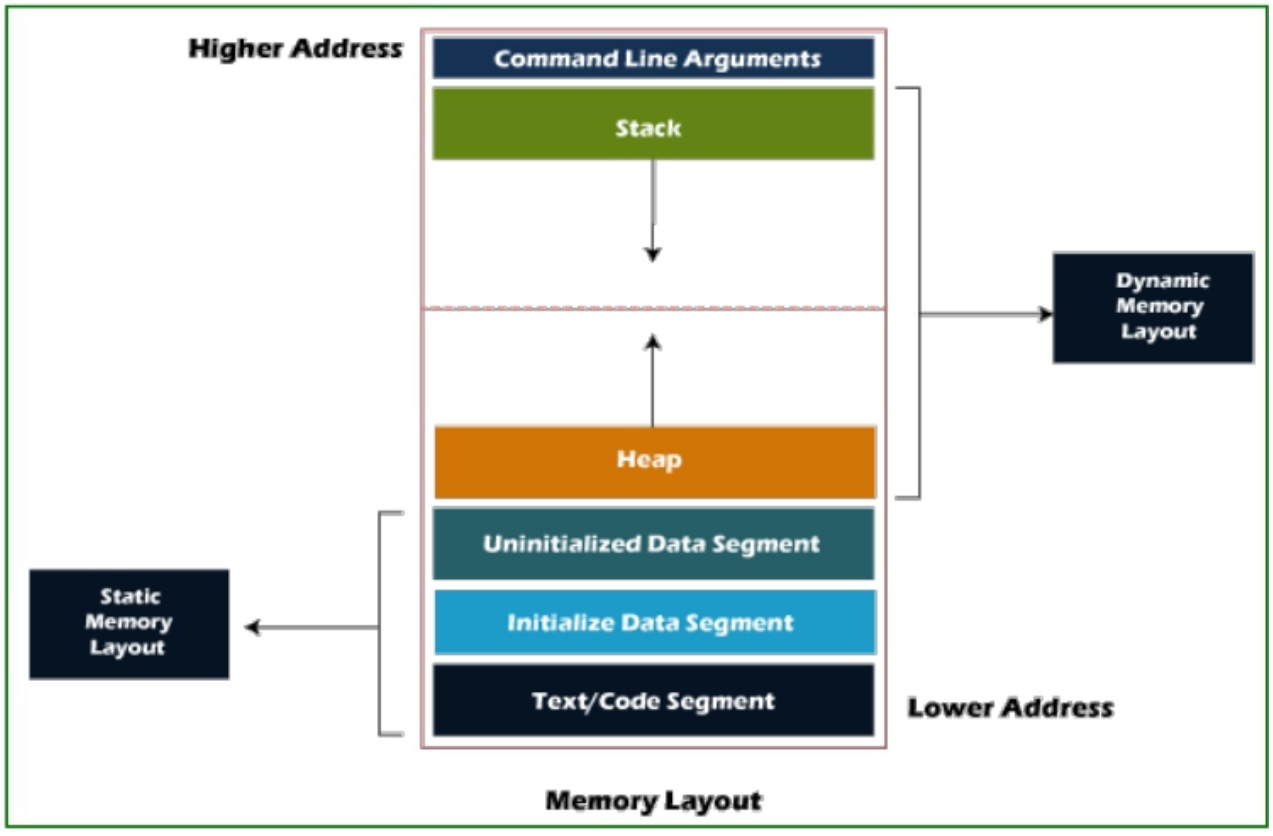
POINTERS & MEMORY NOTES:

- Author: Jude Thaddeau Data
- GitHub: Zero-Luminance
- Sources:
  - Programming, Problem Solving, and Abstraction with C (Alistair Moffat)
  - mycodeschool (YouTube)

MEMORY STRUCTURE:

- 'Random Access Memory' (RAM) is a computer's non-permanent or short term memory used to store things like working data & machine code
- 'Address' are BYTES in memory allocated store information of varying sizes & represented as non-negative 'hexadecimal' numbers
- RAM in the context of programming, stores information such as variables, pointers, etc at addresses, & is divided & ordered in a hierarchy:
  - 'Code Segment' stores executable instructions (e.g: increment, decrement, declaration, etc)
  - 'Static Segment' contains immutable values kept over the lifetime of program execution (e.g. string literals)
  - 'Global Segment' contains global & local variables declared STATIC & is kept over the lifetime of program execution
  - 'Heap Segment' contains REQUESTED & ALLOCATED memory assigned during run time
  - 'Stack Segment' contains memory for the main & function, ADDED & STORED as stacks holding parameters & local variables; removed from stack (deleted from memory) when function completes



- 'Compilation' is the process of converting source code (human readable) into machine code (computer executable)
- 'Static Memory' is FIXED memory allocated upon program compilation
  - Includes: code, static, global, & heap segments
- 'Dynamic Memory' is memory that can be requested then allocated DURING program execution (run time)
  - Includes only: heap

Memory Sizes Of Variable Types:

- 'sizeof(*type*)' returns the number of bytes of a specified type
- NOTE: sizeof function DISTINGUISHES between the sizes of arrays & pointers
- Examples:

C Basic Data Types	32-bit CPU		64-bit CPU	
	Size (bytes)	Range	Size (bytes)	Range
char	1	-128 to 127	1	-128 to 127
short	2	-32,768 to 32,767	2	-32,768 to 32,767
int	4	-2,147,483,648 to 2,147,483,647	4	-2,147,483,648 to 2,147,483,647
long	4	-2,147,483,648 to 2,147,483,647	8	-9,223,372,036,854,775,808-9,223,372,036,854,775,807
long long	8	9,223,372,036,854,775,808-9,223,372,036,854,775,807	8	9,223,372,036,854,775,808-9,223,372,036,854,775,807
float	4	3.4E +/- 38	4	3.4E +/- 38
double	8	1.7E +/- 308	8	1.7E +/- 308

POINTERS:

- 'Pointers' stores the memory address of another variable
- 'Pointer Operators' are symbols put in front of pointer variables to ACCESS, ASSIGN or CHANGE either:
  - '&' for a pointer/regular variable's 'address'
  - ' * ' for pointer/regular variable's 'underlying value', ALSO used for pointer declaration
  - NOTE: when using pointer variables *WITHOUT* operators, use ONLY for the ASSIGNMENT of other variable ADDRESSES or DYNAMIC memory allocation

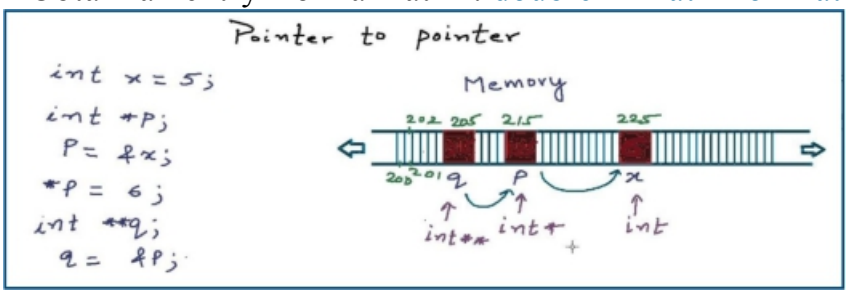
Address	Content	Name	Type	Value
90000000	00	iii	int	000000FF(255_{10})
90000001	00			
90000002	00			
90000003	FF			
90000004	FF	sss	short	FFFF(-1_{10})
90000005	FF			
90000006	1F	ddd	double	1FFFFFFFFFFFFFFF ($4.4501477170144023E-308_{10}$)
90000007	FF			
90000008	FF			
90000009	FF			
9000000A	FF			
9000000B	FF			
9000000C	FF			
9000000D	FF			
9000000E	90	ptr	int*	90000000
9000000F	00			
90000010	00			
90000011	00			

Note: All numbers in hexadecimal

- 'Void Pointers' are pointers without a type & MUST be 'typecasted'
- 'Dereference' is the process of obtaining the address of a data item held in another location from a pointer
- NOTE: pointers can ONLY be assigned to variables or other pointers of the SAME type

Pointers To Pointers & Higher Order Pointers Example Uses:

- Obtain a list of characters (word): char *word or word[]
- Obtain a list of words (sentence): char **sentence or sentence[][]
- Obtain a list of sentences (monologue): char ***monologue or monologue[][][]
- Obtain an entry from a matrix: double **matrix or matrix[][]



Pointers As Function Arguments:

- 'Pointer Function Arguments' or 'Call By Reference' are input parameters that accept the address of supplied arguments
- PURPOSE: provides another mechanism to change & save the underlying value of a variable (once the function call is removed from the stack) without returning
- EXPLANATION: upon a call by reference, the address of the variable is accessed & modified in the static segment rather than the stack memory (which will be deleted upon execution)
- NOTATION:
  - '&' operator for non-pointer arguments
  - NO operators for pointer arguments

Example. A procedure that swaps the value of two variables:

```
void swap(int a, int b) {
    int temp = a ;
    a = b ;
    b = temp ;
}

/* example usage */
int x = 10 ;
int y = 20 ;
swap(x,y) ;
/* x = 10, y = 20 */
```

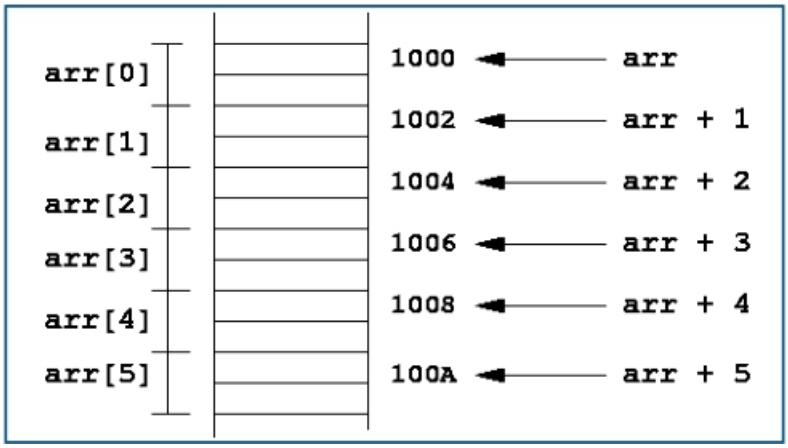
The function has no effect because calling it a *and* b *are copies of* x *and* y. x *and* y *remain unaffected.*

```
void swap(int *a, int *b) {
    int temp = *a ;
    *a = *b ;
    *b = temp ;
}

/* example usage */
int x = 10 ;
int y = 20 ;
swap(&x,&y) ;
/* x = 20, y = 10 */
```

By passing pointers, the function can access the variables x and y in the caller and can swap them.

Arrays & Pointers:

- 'Arrays' are FIXED SIZED contiguous linear collection of same-type variables
- When declared, arrays are simply POINTERS POINTING to the FIRST value in the collection
- '[Square Brackets]' are array subscripts & NOT interpreted differently to ' * ' for pointers; regardless as variables or function arguments
- 'Buddy Variables' accompany arrays (main & function arguments) to keep track of either:
  1) Array Size
  2) Number Of Elements Read
- SIZE DECLARATION: always initialised during compilation & execution (cannot be changed while program is running: see dynamic memory)



NOTE:
Pointer arithmetic can be used to both access the:

1) Address (&) of the element array
2) Underlying values (*) of the element array

Arrays As Function Arguments:

- 'Array Function Arguments' (AFA) are read as pointers to the first address & value in the collection of elements (reduces memory load on stack)
- RULE: functions with AFA should contain a buddy variable
- EFFECT: changes made to values within the passed array/pointer are RETAINED after the function call has finished executing (removed from the stack)

Character Arrays & Pointers:

- 'Strings' are sequences of characters (type char) within an array
- 'Null Byte' ('\0') is a sentinel character ALWAYS added after the sequence of characters & allows the string to be printed as whole via '%s' format descriptor
- Character Array Size = (# of characters) + 1 [for null byte]
- Examples:

```
char s1[5] = {'H', 'e', 'l', 'l', 'o'};
char s2[6] = {'W', 'o', 'r', 'l', 'd', '\0'};
char s3[100] = "Goodbye";
char s4[] = "Pluto";
char *s5 = "Farewell Neptune";
```

NOTE: only s1-s5 are correct array characters

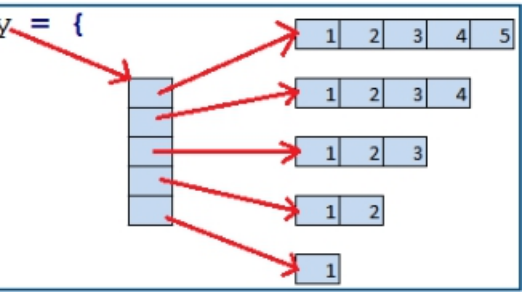
NOTE:
- Declaring s3-s5 automatically places a null byte at the end
- Declaring arrays similar to s3-s5 requires "double quotes" & NOT 'single quotes'

- 'String Constants' (e.g. *s5) are strings declared via pointers & are instead stored in the text/code segment of memory (requires 'const' keyword type AFA)

Pointers & 2D Arrays (Higher Dimensional Arrays):

- 'Two Dimensional Array' is an array of one dimensional arrays
- FORMAT: array[row][column] = value

```
int[][] triangleArray = {
    {1, 2, 3, 4, 5},
    {2, 3, 4, 5},
    {3, 4, 5},
    {4, 5},
    {5},
};
```



NOTE:
The output of a 2D array WITHOUT full subscripts only returns the address of the first element belonging to the first array

- POINTERS: **array
  - **array is a points the first collection of pointers
  - *array points to the first element of items
- When 2D arrays are passed into functions, ONLY the dominant dimension (row) can be left unspecified
- NOTE: it is NOT recommended to use pointer arithmetic for 2D arrays

arr	**Points to 0^{th} 1-D array**
***arr**	**Points to 0^{th} element of 0^{th} 1-D array**
(arr + i)	**Points to i^{th} 1-D array**
***(arr + i)**	**Points to 0^{th} element of i^{th} 1-D array**
***(arr + i) + j)**	**Points to j^{th} element of i^{th} 1-D array**
***(*(arr + i) + j)**	**Reprents the value of j^{th} element of i^{th} 1-D array**

Stack Versus Heap:

Stack:	Heap:
- Upon compilation, the memory size allocated to the code, static/global & stack segments is calculated & does NOT grow throughout the entire program lifecycle - 'Stack Overflow' is the process where the layers of stacks exceeds the maximum amount of memory the stack segment is allowed to hold; program crash - NOT possible to change the scope of variables within the stack - Arrays on the stack need to KNOW it's SIZE upon declaration & CANNOT be changed thereafter - Layers in the stack follow the rule: 'last in first out' (LIFO)	- Arrays/Pointers CAN change the SIZE or the number of elements that can be held - Programmers can control whether memory is either ADDED (malloc, calloc, & realloc) or REMOVED (free) from the heap - 'Heap Overflow' is the process where the amount of memory intended to be dynamically requested exceeds the maximum amount of memory the heap is able to accomodate - NOTE: 'Heap' has a different definition as a TREE-BASED data structure

Stack	Heap
Memory is managed for you	Memory management needs to be done manually
Smaller in Size	Larger in Size
Access is easier and faster and cache friendly	Causes more cache misses because of being dispersed throughout the memory
Not flexible, allotted memory cannot be changed	Flexible and allotted memory can be altered
Faster access, allocation and deallocation	Slower access, allocation and deallocation
Elements' scope is limited to their threads	Elements are globally accessible in the application
OS allocates the stack when the thread is created	OS is called by the language in the runtime to allocate the heap for the application

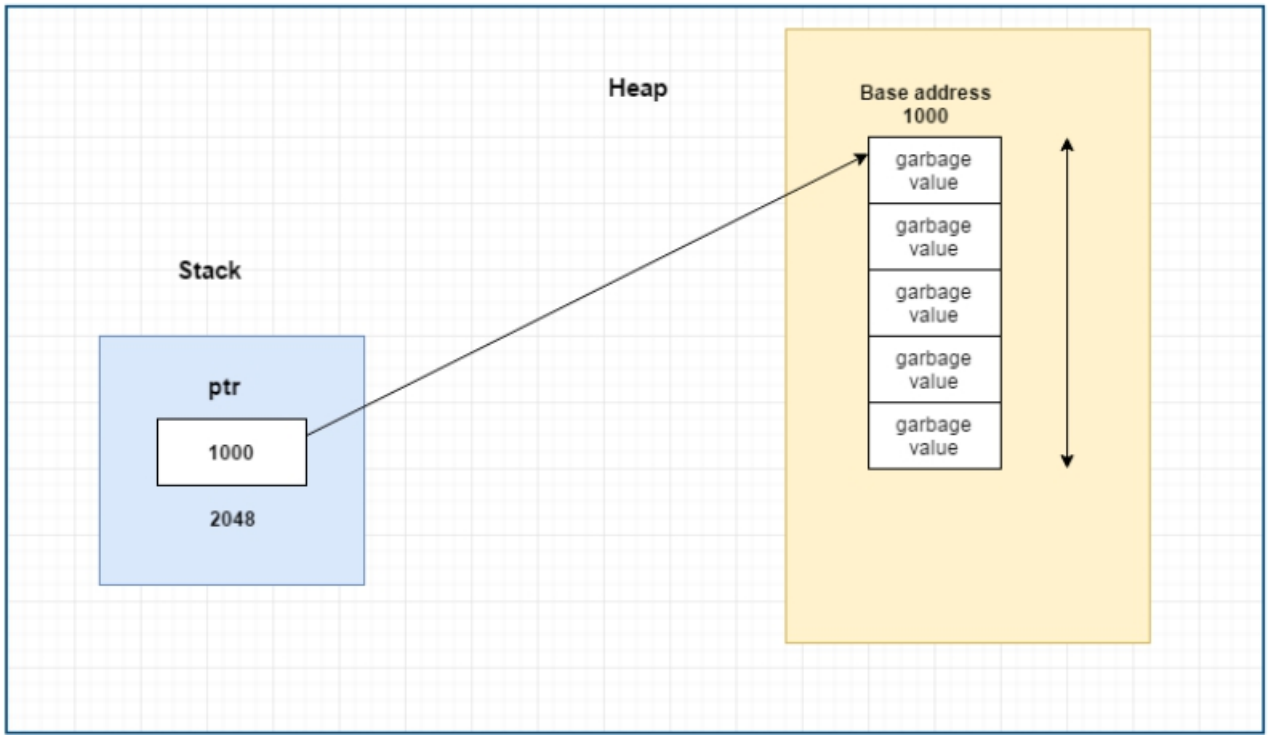
Pointers & Dynamic Memory (Heap):

Allocating Memory:

- 'Dynamic Memory Allocation' functions returns the FIRST ADDRESS from the heap that is able to accommodate the size of requested memory:

(void *)malloc(size_t size)	- Allocates fresh segment of memory of size bytes - Deallocated upon a realloc call
(void *)calloc(size_t nmemb, size_t size)	Similar to malloc except 2 ARGUMENTS are taken & INITIALISES allocated spaces as ZERO
(void *)realloc(void *ptr, size_t size)	Used to change the initial malloc allocation with a SMALLER or LARGER space whilst COPYING contents over to said new space
assert(condition_to_be_upheld)	Can be used to exit program if dynamic memory allocation FAILS

- NOTES:
  - 'NULL' is returned if UNABLE to allocate heap memory
  - 'Number of asterisks' in the dynamic memory allocation MUST MATCH the amount found to the pointer receiving the address
  - 'Type Casting' must be applied to all dynamic memory allocation functions
  - 'Array Subscripts' or 'Pointer Arithmetic' should be used to move between allocated memory spaces
  - For string-based pointers ADD EXTRA BYTE to accommodate NULL
  - When EXPANDING array size via realloc, increase GEOMETRICALLY
  - 'size' is equal to the (number of array elements)*(size of data type) [Specifically for malloc & realloc ONLY]



Deallocating Memory:

- 'Dynamic Memory Deallocation' functions deletes the allocated heap memory:

void free(void *ptr)	Deallocates the segment of memory indicated by ptr & returns it to the pool of available memory

- NOTE: Upon free() function call, pointer STILL POINTS to the same heap address: ASSIGN previously allocated address as NULL

Pointers As Function Returns:

- 'Function Return Pointers' (FRP) returns the address of a dynamically allocated pointer variable
- WARNING:
  - NEVER return the variable address created on the stack (address becomes junk value once deleted from stack)
  - Address returned from the heap call only be deallocated by free (no junk addresses)
- Example uses: linked list data structure functions

```
list_t
*insert_at_head(list_t *list, data_t value) {
        node_t *new;
        new = (node_t*)malloc(sizeof(*new));
        assert(list!=NULL && new!=NULL);
        new->data = value;
        new->next = list->head;
        list->head = new;
        if (list->foot==NULL) {
                /* this is the first insertion into the list */
                list->foot = new;
        }
        return list;
}
```

```
list_t
*get_tail(list_t *list) {
        node_t *oldhead;
        assert(list!=NULL && list->head!=NULL);
        oldhead = list->head;
        list->head = list->head->next;
        if (list->head==NULL) {
                /* the only list node just got deleted */
                list->foot = NULL;
        }
        free(oldhead);
        return list;
}
```

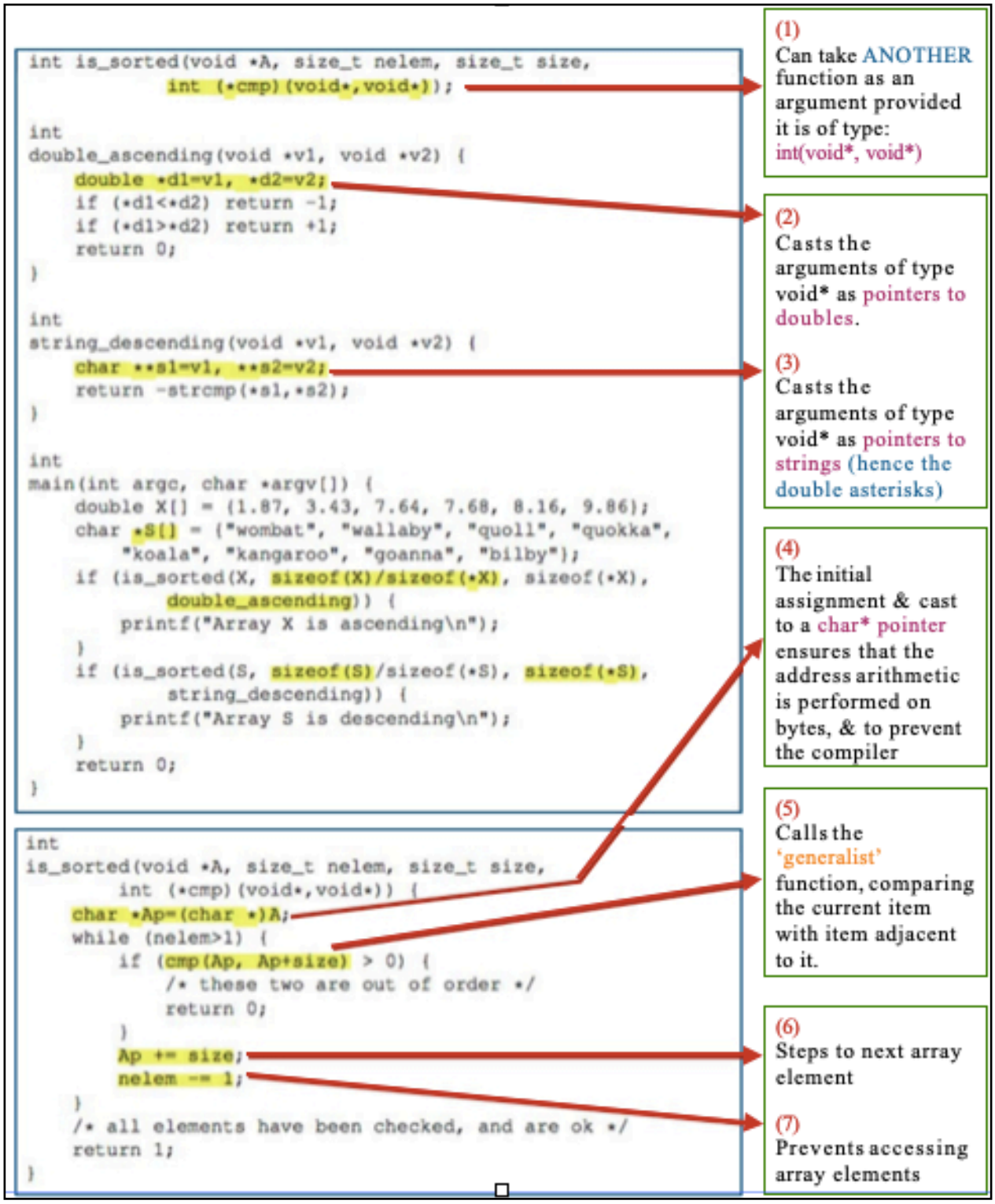
Function Pointers:

- 'Function Pointers' are used to store the address of functions
- SYNTAX:
  1) (*F)(argument list)
  2) F(argument list)
  - 'Argument List' is a collection of the input DATA TYPES
- Code memory segment stores ordered instructions of the way a program will execute; one instruction per address
- 'Function Addresses' are stored as a contiguous blocks in the code/text memory segment (where function pointers point to)
- UTILITY: function pointers can be passed as arguments
- Example:

```c
#include <stdio.h>
// A normal function with an int parameter
// and void return type
void fun(int a)
{
    printf("Value of a is %d\n", a);
}

int main()
{
    // fun_ptr is a pointer to function fun()
    void (*fun_ptr)(int) = &fun;

    /* The above line is equivalent of following two
       void (*fun_ptr)(int);
       fun_ptr = &fun;
    */

    // Invoking fun() using fun_ptr
    (*fun_ptr)(10);

    return 0;
}
```

Polymorphic Function Pointers:

- 'Polymorphic Function Pointers' are functions that utilises function pointers as an input argument
- UTILITY: a polymorphic function can operate on MULTIPLE INPUT DATA TYPES given specific function pointer(s) & void* argument list(s)
- Example 1:
  - Polymorphic Function: is_sorted [checks how array elements are organised]
  - Pointer Function 1: double_ascending [checks if an array of doubles is ascending]
  - Pointer Function 2: string_descending [checks if an array of strings is



- 'const void*' does NOT just provide a polymorphic purpose, but allows the compiler to handle arguments more efficiently
- Example 2:
  - Polymorphic Function: qsort [fast array-sorting algorithm]
  - Pointer Function: string_ascending [checks if array of strings is ascending]

```
int
string_ascending(const void *v1, const void *v2) {
    return strcmp(*(char**)v1,*(char**)v2);
}

int
main(int argc, char *argv[]) {
    int i;
    char *S[] = {"koala", "kangaroo", "quoll", "quokka",
        "wombat", "goanna", "wallaby", "bilby"};
    qsort(S, sizeof(S)/sizeof(*S), sizeof(*S),
            string_ascending);
    for (i=0; i<sizeof(S)/sizeof(*S); i++) {
        printf("%s\n", S[i]);
    }
    return 0;
}
```

(1) Type casting void inputs into strings & then accessing their value

```
bilby
goanna
kangaroo
koala
quokka
quoll
wallaby
wombat
```

Constant Quantifier:

- 'const' is data type added either in-front or behind (both optional) traditional datatypes & specifies that casted variables of this type WILL NOT change
- 'Pointer To Constant' only allows the pointer to change the address it is pointing to, but nor the underlying value found at the address
  - Examples: const int *ptr OR int const *ptr
- 'Constant Pointer To Variable' only allows the pointer to change the underlying value at the address, but the pointer may not reassigned to point elsewhere
  - Example: int *const ptr
- 'Constant Pointer To Constant' prevents the pointer from being able to point to a different address & prevents change to the underlying value of current address
  - Example: const int *const ptr
- USE CASES:

1) Use in function parameters passed by reference where the function DOES NOT modify or free the data pointed to

```
int find(const int *data, size_t size, int value);
```

2) Use for constants that might otherwise be defining using #define

```
const double PI = 3.14;
```

3) Never use in a function PROTOTYPE for a parameter passed by value (non-pointer based arguments)

```
// don't add const to 'value' or 'size'
int find(const int *data, size_t size, int value);
```

4) Where appropriate, use 'const volatile' on locations that cannot be changed by the program but still change

```
const volatile int32_t *DEVICE_STATUS =  (int32_t*) 0x100;
```

5) [optional] The parameters to a function within a function IMPLEMENTATION can be marked as cons

```
int function_a(char * str, int n)
{
    ...
    function_b(str);
    ...
}
```

6) [optional] Function return values or calculations that are obtained & then never change

```
const char str[] = "hello world\n";
char *s = strchr(str, '\n');
*s = '\0';
```

Memory Leaks:

- 'Memory Leak' occurs when a program incorrectly manages memory allocations in such a way that memory that is NO LONGER NEEDED is NOT RELEASED
- In C/C++, FAILURE to use free() for growing number of malloc, calloc & realloc calls results in a the heap memory storing more information
- Once the heap reaches & GOES OVER the maximum amount of memory it can accommodate, the program crashes