3. Dynamic Memory Allocation

Memory allocation

When memory is dynamically allocated, it is allocated in heap memory. Such memory has to be allocated and deallocated explicitly by the programmer, as opposed to stack memory which is temporary and is automatically deallocated when a function returns.

`malloc`

The prototype for `malloc` is `void * malloc(size_t size);`

`malloc` allocates `size` bytes of memory and returns a `void` pointer to the allocated memory.

In C89, a `void` pointer needs to be explicitly cast to the appropriate type. In C99, a `void` pointer can be cast implicitly.

If `malloc` can't locate a large enough block of memory, it returns a null pointer and sets errno to `ENOMEM`.

Attempting to access memory through a null pointer results in undefined behaviour.

Memory Leak

Not deallocating memory allocated by `malloc` can lead to memory leaks.

```
#include <stdlib.h>

int allocate() {
        int *pt;
        if ((pt = (int *) malloc(sizeof(int))) == NULL) return(-1);
        return(0);
}

int main() {
        return(allocate());
}
```

Here, the pointer `pt` is stored in the stack of `void allocate()` but it points to memory stored in the heap. Since stack memory is deallocated after a function terminated, the pointer `pt` will be deallocated. However, the memory assigned by `malloc` is still allocated but cannot be accessed anymore since the pointer to is lost.

If enough memory is leaked, the program will run out of memory and encounter an out of memory error, or ENOMEM.

`free`

The prototype for `free` is `void free(void *ptr);`

`free` is used for deallocating allocations that were created via the preceding allocation functions. `free` deallocates the memory allocation pointed to by `ptr`. `free` has no return value.

If `ptr` is a NULL pointer, no operation is performed.

The memory leak in the previous example can now be fixed as follows

```
int allocate() {
        int *pt;
        if ((pt = (int *) malloc(sizeof(int))) == NULL) return(-1);
        free(pt);
        return(0);
}

int main() {
        return(allocate());
}
```

`free` does not reset the pointer or value at the address the pointer points to, it simply indicates to the memory management system that the memory can be allocated again. Pointers that point to `free`'d memory are called dangling pointers. Pointers should be set to null after being freed to avoid dereferencing deallocated memory.

Each call to `malloc` needs to be `free`'d individually. In nested data structures, care should be taken to determine the order in which `free` is called so as to not use a dangling pointer.

```
int **pointers = malloc(sizeof(int *) * 2);
// The first element is a pointer to an int
pointers[0] = malloc(sizeof(int));
```

```
// The second element is a pointer to an array of ints
pointers[1] = malloc(sizeof(int) * 3);
free(pointers[0]);
free(pointers[1]);
free(pointers);
```

`calloc`

The prototype for `calloc` is `void * calloc(size_t count, size_t size);`

`calloc` contiguously allocates enough space for count objects that are size bytes of memory each and returns a pointer to the allocated memory.

The allocated memory is filled with bytes of value `0`.

If there is an error, `calloc` returns a null pointer and sets errno to `ENOMEM`.

`realloc`

The prototype for `realloc` is `void * realloc(void *ptr, size_t size);`

`realloc` tries to change the size of the allocation pointed to by `ptr` to `size`, and returns `ptr`.

If there is not enough room to enlarge the memory allocation pointed to by `ptr`, `realloc` creates a new allocation, copies as much of the old data pointed to by `ptr` as will fit to the new allocation, frees the old allocation, and returns a pointer to the allocated memory.

If `ptr` is NULL, `realloc` is identical to a call to `malloc` for `size` bytes.

If `size` is `0` and `ptr` is not NULL, a new, minimum sized object is allocated and the original object is freed.

When extending a region allocated with `calloc`, `realloc` does not guarantee that the additional memory is also `0`-filled.

Passing a pointer to `realloc` that is not NULL and didn't come from a previous call to `malloc`, `calloc`, or `realloc` results in undefined behaviour.

If `realloc` can't locate a large enough block of memory, it returns a NULL pointer and sets errno to `ENOMEM`. The input pointer is still valid if reallocation failed.
