Dynamic Memory Allocation

Variable Scope and Lifetime

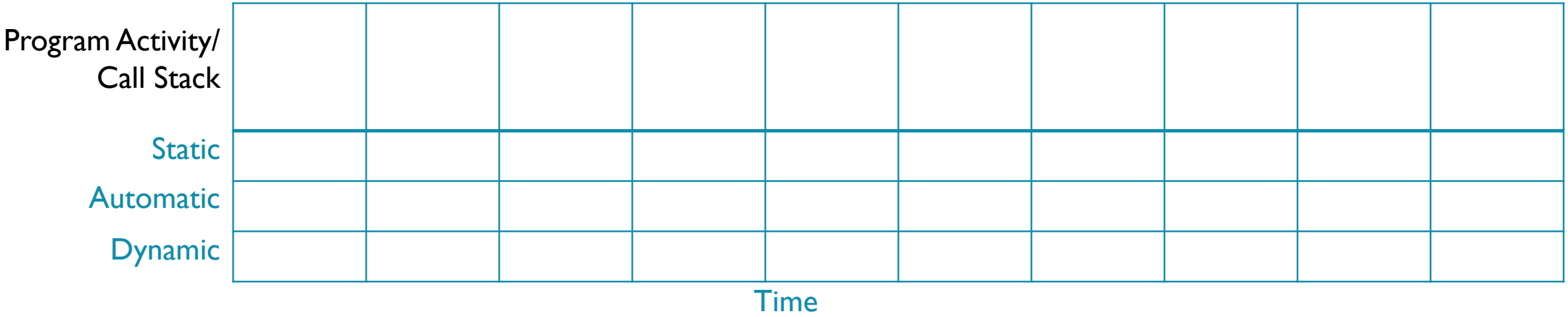
- Scope: What parts of program can access the variable?
  - File: Accessible to everything in **file** after declaration
  - Block: Accessible to everything in **block** after declaration
    - Block defined by matching { and }
    - A function is a block
    - Nested blocks are possible (e.g. automatic_block_scope)
- Lifetime: How long does the variable exist?
  - Static: lifetime = program. Memory not reusable
  - Automatic: lifetime = block. Memory reusable
  - Dynamic: lifetime = user-defined. Memory reusable

```
int static_file_scope;

void function1(int arg) {
    int automatic_function_scope;
    static int static_function_scope;
    if (arg > 0) {
        int automatic_block_scope;
    }
}
```

- Is variable allocated space in memory?
  - Variables allocated space in memory by default
  - Variables with least scope and lifetime are easier for compiler to optimize by eliminating memory, just using registers

Variable Lifetimes



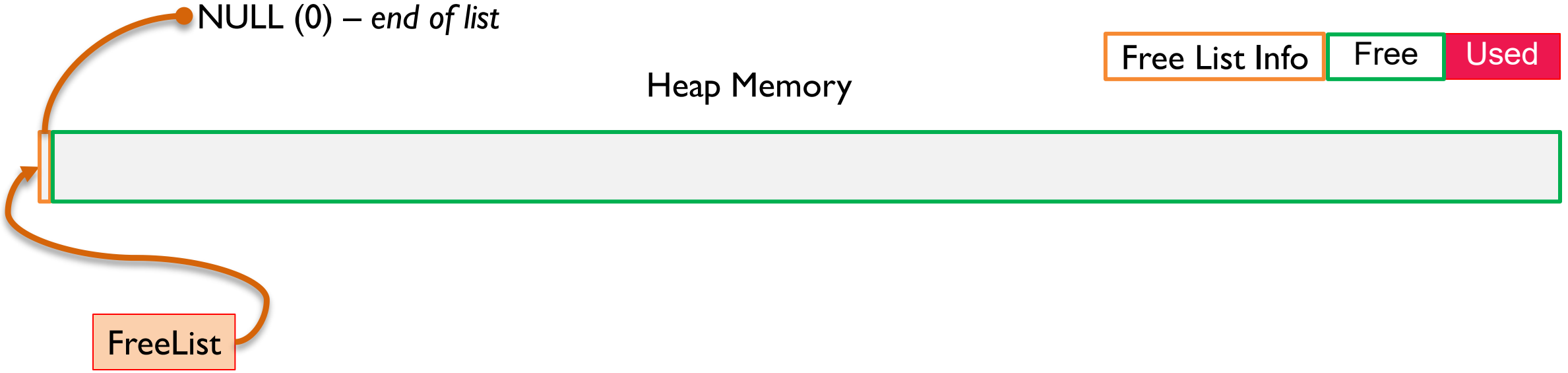
- Static: lifetime = program
  - Allocated (given address) at build time, never freed. Not reusable
- Automatic function: lifetime = function
  - Allocated at function entry, freed at function exit. Reusable
- Automatic block: lifetime = block
  - Allocated at { block entry, freed at } block exit. Reusable
- Dynamically-allocated: lifetime = from allocate to free
  - Allocated by user, freed by user
  - Lifetime not limited to program , function or block
  - Reusable
- Further information:
  - https://blog.feabhas.com/2010/09/scope-and-lifetime-of-variables-in-c/

Dynamic Memory Allocation Functions in C

Standard library dynamic memory allocation functions <stdlib.h>

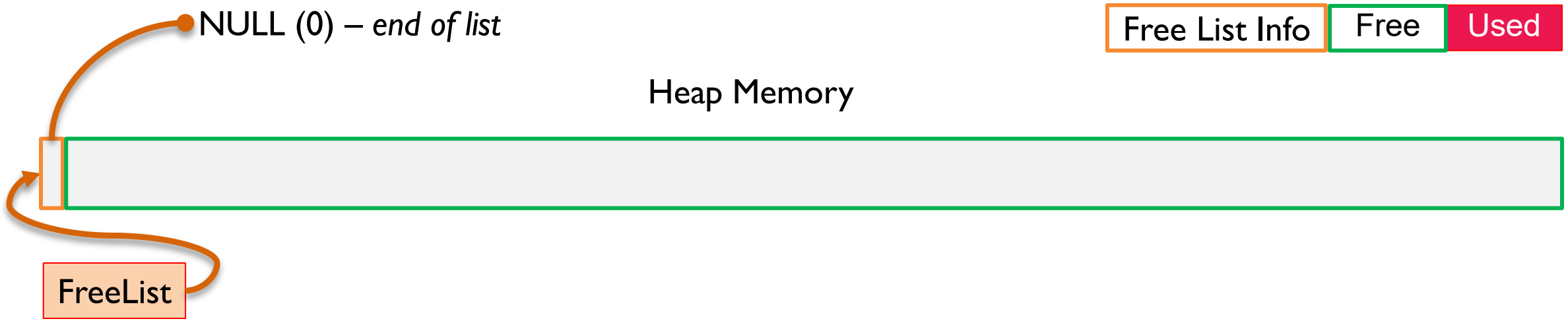
- void * **malloc**(size_t **size**)
  - Allocate memory block of **size** bytes
- C operator **sizeof**() give size in bytes
  - Of data type, variable or expression result
- void * **calloc**(size_t **nitems**, size_t **size**)
  - Allocate memory block of **nitems*size** bytes and clear it to zero
- void **free** (void * **ptr**)
  - Return allocated block of memory for reuse
- void * **realloc**(void * **ptr**, size_t **size**)
  - Change size of previously allocated block. Shrink existing block, or allocate a new block and copy over data (limited by smaller size)

How Does Dynamic Memory Allocation Work?



- Uses the "heap" section of memory
  - Heap_Size defined in startup_MKL25Z4.h
  - Statically allocated at build time
- Keep track of free space with linked list starting with FreeList pointer
  - Each list entry holds size of free memory block, pointer to next free memory block

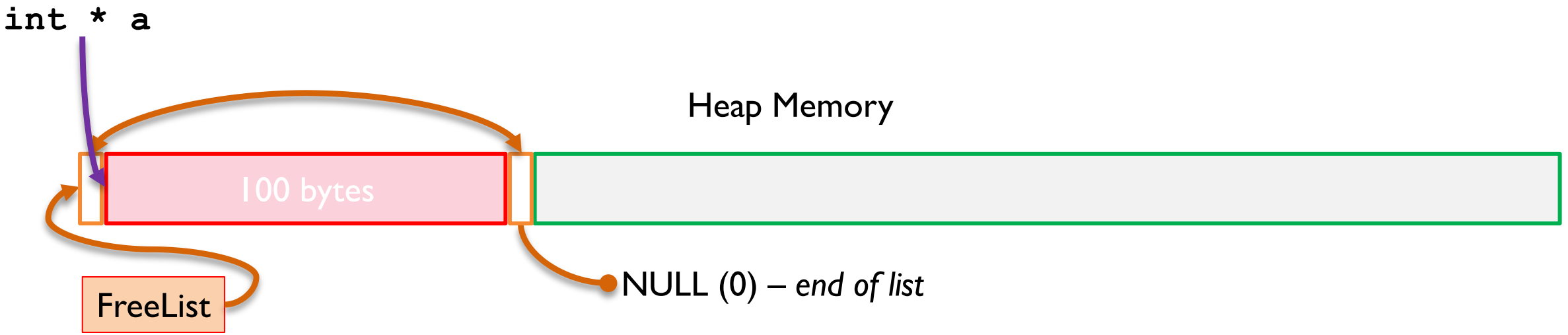
Example Operations



- Allocation of N byte block (malloc, calloc)
  - Find first block B which is large enough
  - If size of B > N bytes, split B into two blocks (one used, one unused)
  - Update list pointers as needed
  - Return pointer to start of allocated block

```
int * a, * b, * c;
a = (int *) malloc(100);
b = (int *) malloc(200);
free(a);
a = (int *) 0;
c = (int *) malloc(64);
```

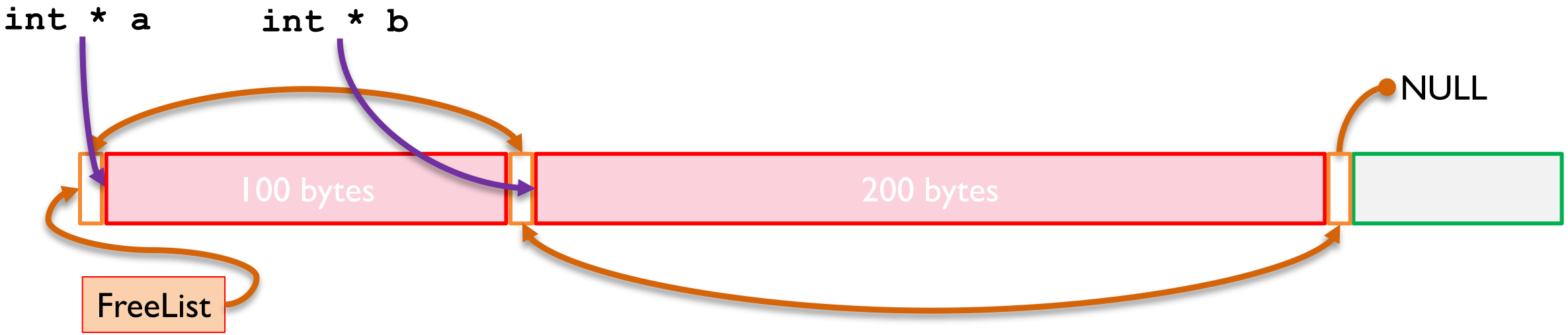
Allocate 100-Byte Block for Pointer a



- Allocation of N byte block (malloc, calloc)
  - Find first block B which is large enough
  - If size of B > N bytes, split B into two blocks (one used, one unused)
  - Update list pointers as needed
  - Return pointer to start of allocated block

```
int * a, * b, * c;
a = (int *) malloc(100);
b = (int *) malloc(200);
free(a);
a = (int *) 0;
c = (int *) malloc(64);
```

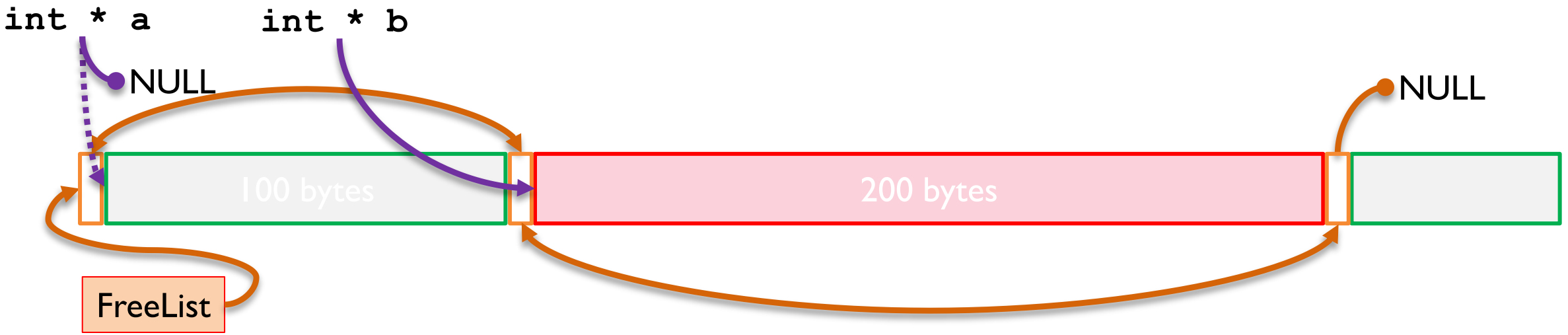
Allocate 200-Byte Block for Pointer b



- Allocation of N byte block (malloc, calloc)
  - Find first block B which is large enough
  - If size of B > N bytes, split B into two blocks (one used, one unused)
  - Update list pointers as needed
  - Return pointer to start of allocated block

```
int * a, * b, * c;
a = (int *) malloc(100);
b = (int *) malloc(200);
free(a);
a = (int *) 0;
c = (int *) malloc(64);
```

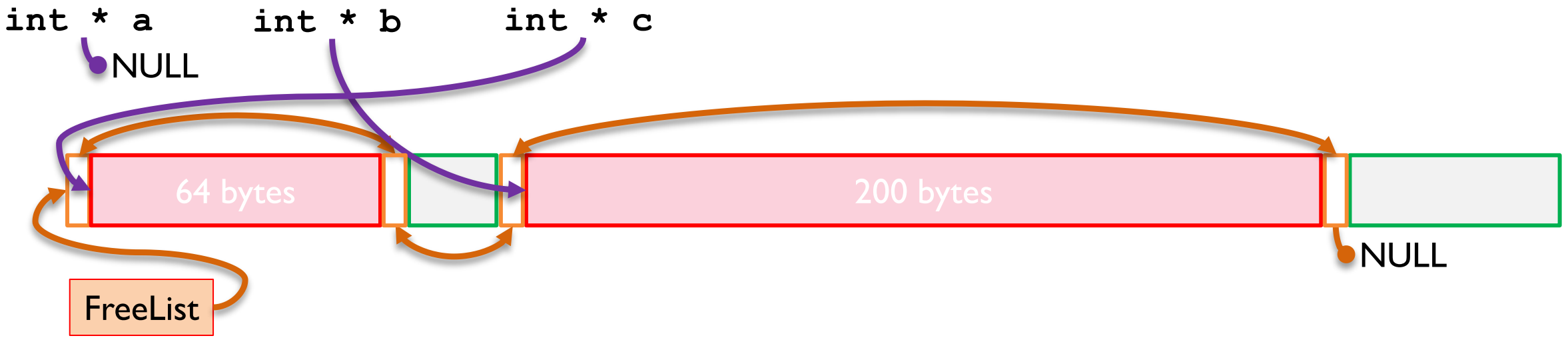
De-Allocate Pointer a's Block



- Deallocation of block (free)
  - If block to free is adjacent to a block in FreeList, merge them and update the list entry
  - Else add list entry with pointer to B (and size) to FreeList
- Note that a is still pointing to the memory block!
  - For safety should set a to null pointer: (int *) 0;

```
int * a, * b, * c;
a = (int *) malloc(100);
b = (int *) malloc(200);
free(a);
a = (int *) 0;
c = (int *) malloc(64);
```

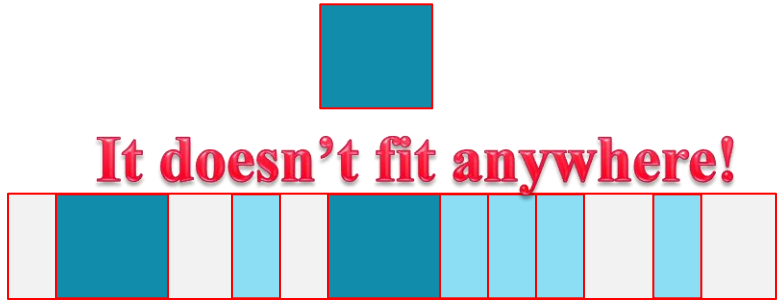
Allocate 64-Byte Block for Pointer c



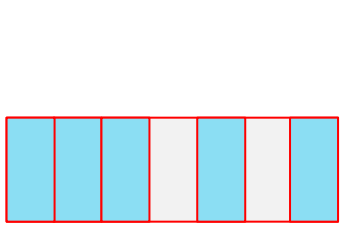
- Allocation of N byte block (malloc, calloc)
  - Find first block B which is large enough
  - If size of B > N bytes, split B into two blocks (one used, one unused)
  - Update list pointers as needed
  - Return pointer to start of allocated block

```
int * a, * b, * c;
a = (int *) malloc(100);
b = (int *) malloc(200);
free(a);
a = (int *) 0;
c = (int *) malloc(64);
```

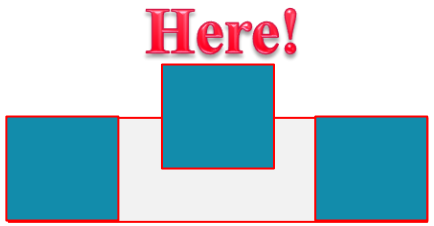
Fragmentation and Dynamic Memory Allocation



Single pool handling all request sizes
(e.g. 16 and 32 bytes)



Pool for 16-byte requests

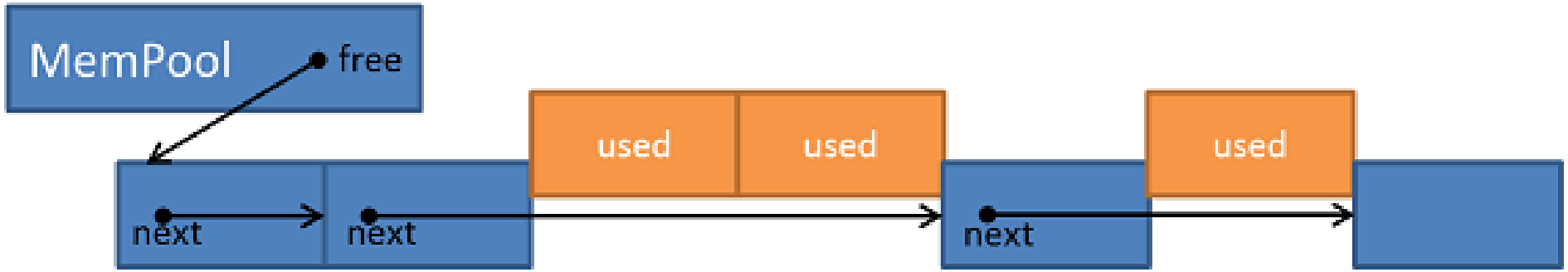


Pool for 32-byte requests

- Standard dynamic memory allocation
  - malloc, calloc, free, etc.
  - Use a single memory pool
- Problem
  - Allocating and freeing different size objects causes "**internal fragmentation**"
  - Free memory space is distributed in fragments which are too small to use

- Custom dynamic memory allocation
  - Use a separate memory pool for each size of data to be allocated
  - Eliminates fragmentation within a pool (internal)
  - Improves timing performance as well
  - Common RTOS feature
- Drawback: fixed partition
  - May have external fragmentation if pools are not sized well for memory requirements

CMSIS-RTOS2 Memory Pool



- Pool contains multiple fixed-size blocks of memory
- Linked list of available memory blocks
- Since all blocks in pool are same size, allocation and freeing are very fast and easy
- Can create multiple pools, one per object (block) size
  - https://www.keil.com/pack/doc/CMSIS/RTOS2/html/group__CMSIS__RTOS__PoolMgmt.html

CMSIS-RTOS2 Memory Pool Creation and Destruction

- osMemoryPoolId_t **osMemoryPoolNew**(block_count, block_size, attributes)
  - Creates and initializes a memory pool
  - Returns ID for pool

- osMemoryPoolId_t **osMemoryPoolDelete**(pool_id)
  - Deletes memory pool, frees up space for other calls to osMemoryPoolNew

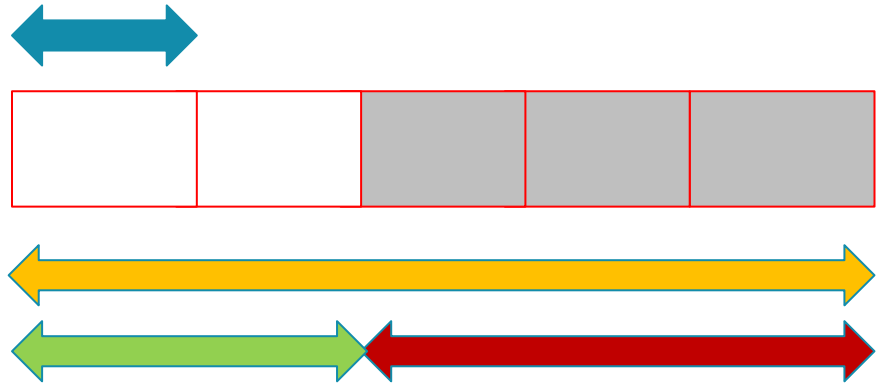
CMSIS-RTOS2 Memory Pool Use

- void * **osMemoryPoolAlloc**(pool_id, timeout)
  - Returns address of allocated memory block, or NULL if no memory available
  - If no memory available, can block until available (or time out)

- osStatus_t **osMemoryPoolFree**(pool_id, block)
  - Frees block in pool for future allocation
  - Return value
    - osOK, osErrorResource, osErrorParameter

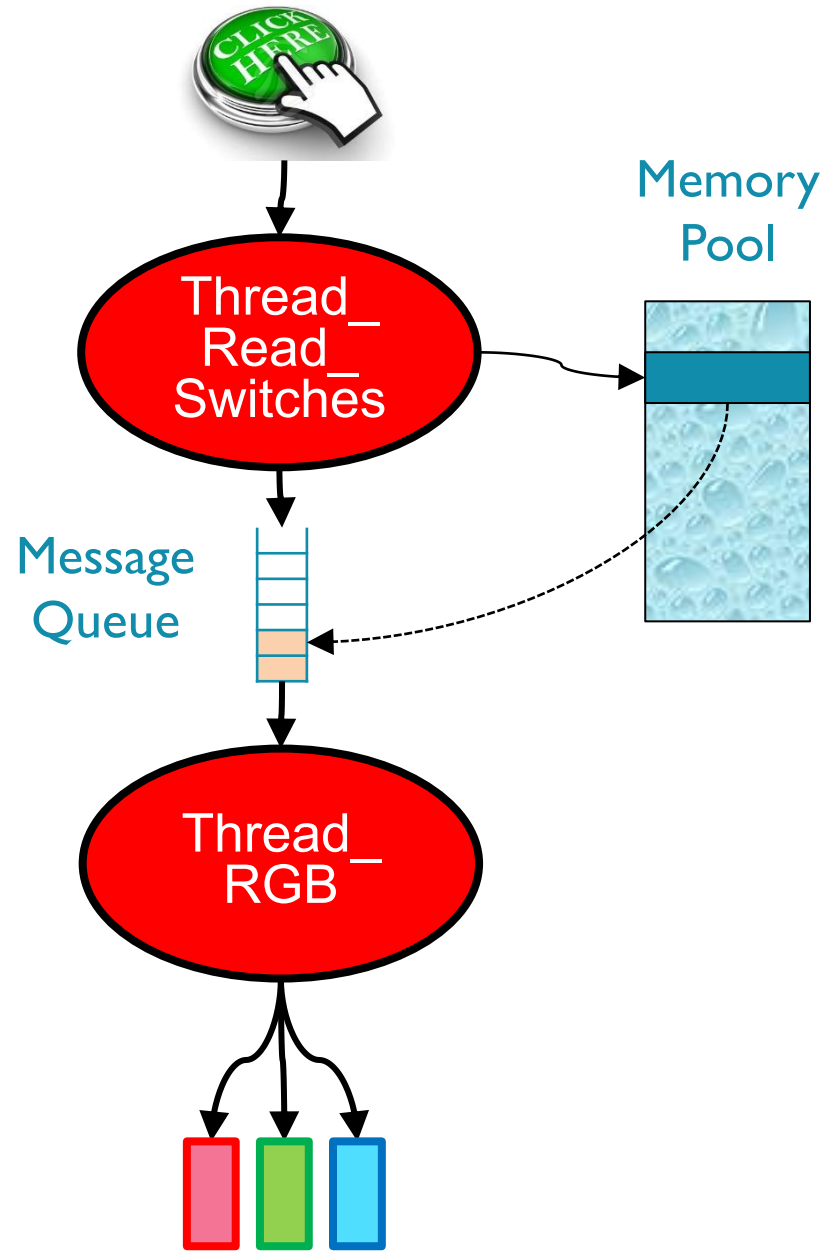
CMSIS-RTOS2 Memory Pool Metrics

- uint32_t **osMemoryPoolGetBlockSize**(pool_id)
  - Returns size of block in bytes
- uint32_t **osMemoryPoolGetCapacity**(pool_id)
  - Returns max. number of memory blocks in pool
- uint32_t **osMemoryPoolGetSpace**(pool_id)
  - Returns number of memory blocks available
- uint32_t **osMemoryPoolGetCount**(pool_id)
  - Returns number of memory blocks used



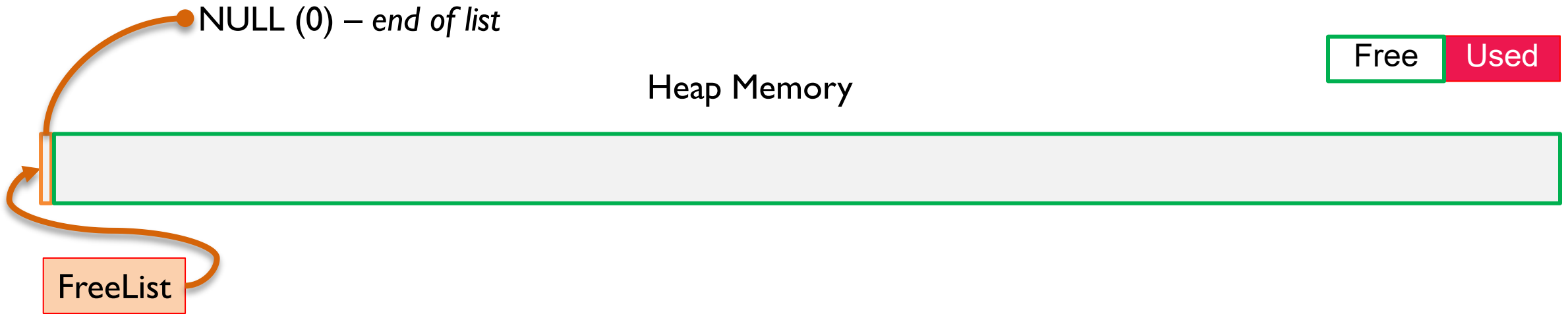
RTX Demo: Memory Pool

- Thread_Read_Switches polls switches
  - If SW1 pressed, start creating an LED flashing pattern in memory
    - Allocates pattern memory from pool
  - When SW1 released, send that pattern as a message to Thread_RGB
    - **Put** copies pattern into message
    - Then free pattern memory, returning it to pool
- Thread_RGB waits for message
  - Plays RGB sequence specified in message



OLD

Example Operations



- Allocation of N byte block (malloc, calloc)
  - Find first block B which is large enough
  - If size of B > N bytes, split B. Update list entry with (size of B)-N
  - Update list pointers as needed
  - Return pointer to start of allocated block
- Deallocation of block (free)
  - If block to free is adjacent to a block in FreeList, merge them and update the list entry
  - Else add list entry with pointer to B (and size) to FreeList

```
int * a, * b, * c;
a = (int *) malloc(100);
b = (int *) malloc(200);
free(a);
c = (int *) malloc(64);
```