In C, memory is divided into two main areas: the stack and the heap.

The Stack

The stack is a region of memory that stores local variables, function parameters, and return addresses. When a function is called, a block of memory is allocated on the stack to store the function's local variables and parameters. This block of memory is called a stack frame.

The stack is a Last-In-First-Out (LIFO) data structure, meaning that the most recently allocated block of memory is the first one to be deallocated. When a function returns, its stack frame is automatically deallocated, and the memory is freed.

Variables allocated on the stack have a limited scope, meaning they are only accessible within the function that allocated them. When the function returns, the variables are no longer accessible.

The Heap

The heap is a region of memory that is used for dynamic memory allocation. When you use `malloc`, `calloc`, or `realloc` to allocate memory, you are allocating memory on the heap.

The heap is a dynamic data structure, meaning that memory can be allocated and deallocated at runtime. When you allocate memory on the heap, you are responsible for freeing it when you're done using it.

Variables allocated on the heap have a global scope, meaning they can be accessed from anywhere in the program.

Why only dynamically allocated memory needs to be freed

When you allocate memory on the stack, it is automatically deallocated when the function returns. However, when you allocate memory on the heap using `malloc`, `calloc`, or `realloc`, you are responsible for freeing it when you're done using it.

If you don't free dynamically allocated memory, it will remain allocated until the program terminates. This can lead to memory leaks, which can cause your program to consume increasing amounts of memory over time.

In contrast, memory allocated on the stack is automatically deallocated when the function returns, so you don't need to worry about freeing it.

Example

Here's an example to illustrate the difference:

```
c
CopyInsert
```

```c
int foo() {
    int x = 10; // allocated on the stack
    int* y = malloc(sizeof(int)); // allocated on the heap
    *y = 20;
    return 0;
}
```

In this example, `x` is allocated on the stack, so it is automatically deallocated when the function returns. `y`, on the other hand, is allocated on the heap, so it needs to be freed using `free(y)` to avoid a memory leak.

I hope this helps clarify things! Let me know if you have any further questions.