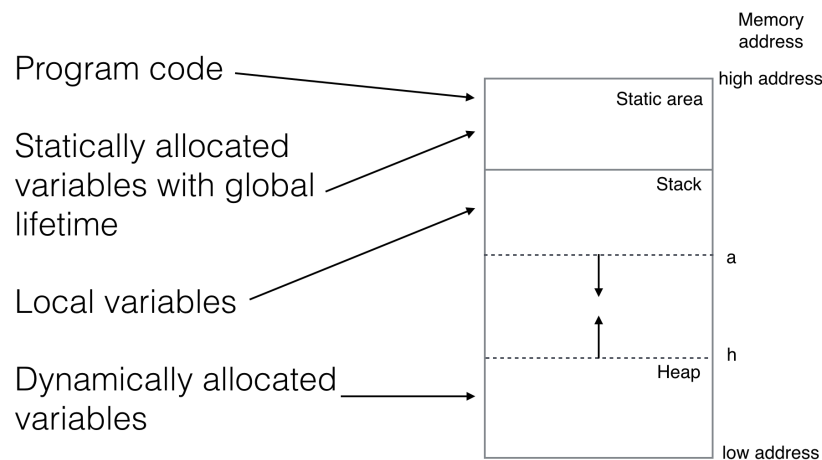


Memory Management (I)

Composition

- The memory of a program consists of **four types of information**
 - **Program code**
 - Statically allocated **variable with global lifetime**
 - **Local variables** and **function calls**
 - **Dynamically allocated variables**
- The **memory is composed of a static area, a stack, and a heap**.
 - Although the stack is implemented as the stack data structure, but the **heap has nothing relevant to the heap data structure**. It is confusing especially if you know data structure. So it is better to clarify here heap in memory is not heap in data structure
- Generally, the **stack is in high memory** and grows down towards the heaps. The **heap is in low memory** and grows up towards the stack.
- In this figure, static area is on top of stack. But depending on the machine architecture, **static area can be either on top of stack or after heap**.



- Program code and statically allocated global variable are in the static area.
- Local variables and function calls are stored in the stack.
- Dynamically allocated variables are stored in heap.
- Exercise
 - Where are the variables, g, l, i, and a in the following program stored in memory?
 - g in static area
 - l, i, and **a are in stack**
 - **the space referred by a using malloc are stored in the stack.**

```

#include <stdio.h>
#include <stdlib.h>

char g = 0xA; // '\n' in ASCII

int main () {
    int l, i, *a;

    printf("Enter the number of integers: ");
    scanf("%d", &l);
    a = (int*) malloc (l * sizeof(int));

    if (a == NULL) {
        printf("Error! memory not allocated");
        exit(0);
    }

    printf("Enter integers: ");
    for (i = 0; i < l; i++) {
        scanf("%d", a+i);
    }

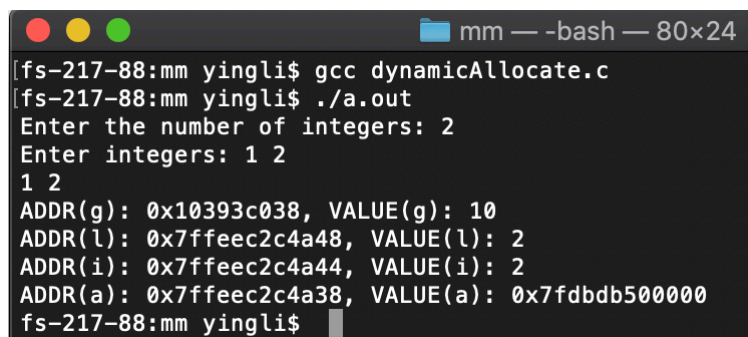
    for (i = 0; i < l; i++) {
        printf("%d ", *a+i);
    }

    printf("%c", g);

    printf("ADDR(g): %p, VALUE(g): %d \n", &g, g);
    printf("ADDR(l): %p, VALUE(l): %d \n", &l, l);
    printf("ADDR(i): %p, VALUE(i): %d \n", &i, i);
    printf("ADDR(a): %p, VALUE(a): %p \n", &a, a);

    free(a);
    return 0;
}

```



```

mm — -bash — 80x24
[fs-217-88:mm yingli$ gcc dynamicAllocate.c
[fs-217-88:mm yingli$ ./a.out
Enter the number of integers: 2
Enter integers: 1 2
1 2
ADDR(g): 0x10393c038, VALUE(g): 10
ADDR(l): 0x7ffec2c4a48, VALUE(l): 2
ADDR(i): 0x7ffec2c4a44, VALUE(i): 2
ADDR(a): 0x7ffec2c4a38, VALUE(a): 0x7ffdbdb50000
fs-217-88:mm yingli$

```

- Demonstrate that a is in stack by adding the two lines of code after free(a), and run.

```

#include <stdio.h>
#include <stdlib.h>

char g = 0xA; // '\n' in ASCII

int main () {
    int l, i, *a;

    printf("Enter the number of integers: ");
    scanf("%d", &l);
    a = (int*) malloc (l * sizeof(int));

    if (a == NULL) {
        printf("Error! memory not allocated");
        exit(0);
    }

    printf("Enter integers: ");
    for (i = 0; i < l; i++) {
        scanf("%d", a+i);
    }

    for (i = 0; i < l; i++) {
        printf("%d ", *a+i);
    }

    printf("%c", g);

    printf("ADDR(g): %p, VALUE(g): %d \n", &g, g);
    printf("ADDR(l): %p, VALUE(l): %d \n", &l, l);
    printf("ADDR(i): %p, VALUE(i): %d \n", &i, i);
    printf("ADDR(a): %p, VALUE(a): %p \n", &a, a);

    free(a);

    a = &l;
    printf("%d\n", *a);

    return 0;
}

```

```

mm — -bash — 80x24
[fs-217-88:mm yingli$ gcc dynamicAllocate.c
[fs-217-88:mm yingli$ ./a.out
Enter the number of integers: 2
Enter integers: 1 2
1 2
ADDR(g): 0x109f4c038, VALUE(g): 10
ADDR(l): 0x7ffee5cb4a48, VALUE(l): 2
ADDR(i): 0x7ffee5cb4a44, VALUE(i): 2
ADDR(a): 0x7ffee5cb4a38, VALUE(a): 0x7fd5b2500000
2
fs-217-88:mm yingli$

```

- If `a` is in heap, after invoking `free(a)`, it should be invalid to assign a value to `a` or print its value. However, `a = &l` and `printf("%d\n", *a)` are valid statements here, which demonstrate that `a` is in stack instead of in heap.
- Besides, comparing `a`'s address and `a`'s value, it's obvious that `a` is in stack and the memory referenced by the address stored in `a` is in heap.