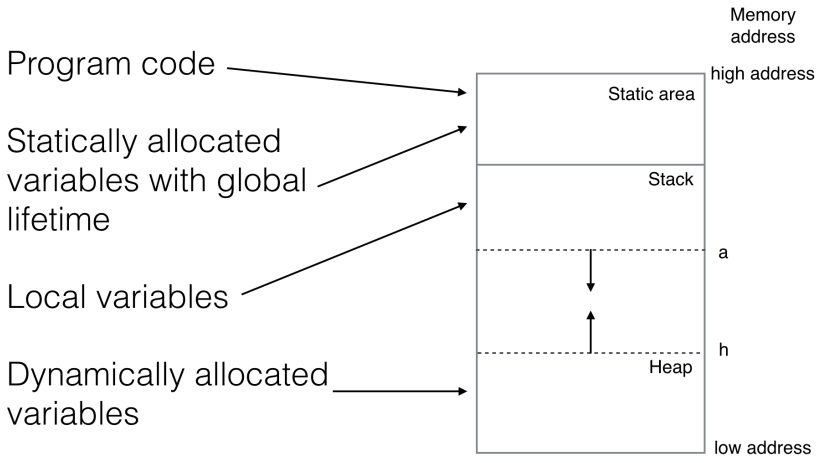
Memory Management (I)

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.**

```c
#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 — 80×24
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): 0x7ffeec2c4a48, VALUE(l): 2
ADDR(i): 0x7ffeec2c4a44, VALUE(i): 2
ADDR(a): 0x7ffeec2c4a38, VALUE(a): 0x7fdbdb500000
fs-217-88:mm yingli$
```

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

```c
#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 — 80×24
[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.

- Now, we know that local variables use stack space, dynamically allocated variables ask space from heap, and statically allocated global variables and programs use static area.
- So, note that use global variables with caution (avoid using too much static area space)

- Each program has fixed size of stack, although the actual size of stack frame is dynamic.
- Each function call push a stack frame into the stack for information like return address, return value, arguments, local variables, copies of registers (e.g., pointer to the address of the function in the program)

- Therefore, a recursive function may need more than one stack frame, and deep recursive functions will need much more stack space.
- From the perspective of memory complexity and space complexity, recursive functions can be expensive.
- Depending on the compiler, deep recursion may cause stack overflow. Even it won't, the program may be inefficient.
- Show fibonacci.c. see the difference between n = 5 and n = 50

- The following fibonacci.c implements fibonacci sequence in two ways: recursion and loop.
- Note: recursive functions can be implement using loops.

```c
#include <stdio.h>
#include "my_timing.h"

/* recursive method: double recursion, variable n grows exponentially */
unsigned long FibonacciR (unsigned n) {
     if (n > 2)
          return FibonacciR(n-1) + FibonacciR(n-2);
     else
          return 1;
}

/* without recursion */
unsigned long Fibonacci (unsigned n) {
     if (n <= 1) {
          return n;
     }

     unsigned long curr = 1;
     unsigned long prev = 1;
     for (int i = 2; i < n; ++i) {
          unsigned long temp = curr;
          curr += prev;
          prev = temp;
     }

     return curr;
}
```

```c
int main () {
     double t1, t2, t3;
     unsigned d;
     while (scanf("%d", &d) != 0) {
          printf("Entered: %d\n", d);
          t1 = get_time_sec();
          unsigned i = FibonacciR(d);
          t2 = get_time_sec();
          unsigned j = Fibonacci(d);
          t3 = get_time_sec();
          printf("FibonacciR number is: %u, Time: %f sec\n", i, t2 - t1);
          printf("Fibonacci number is: %u, Time: %f sec\n", i, t3 - t2);
     }

     return 0;
}
```

- Below are the my_timing.h and corresponding my_timing.c files used in fibonacci.c.

```c
#ifndef MY_TIMING_H

#define MY_TIMING_H

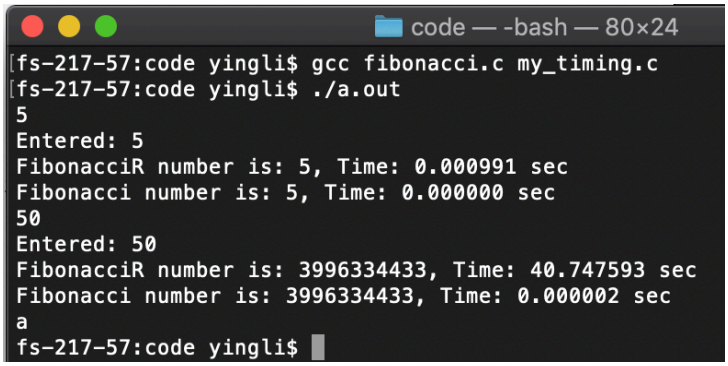
// Return the time in seconds
// Note: it doesn't work to return a float - it must be a double.
double get_time_sec();

#endif
```

```c
#include <sys/time.h> // for get_time_sec
#include <time.h>

// Return the time in seconds
// Note: it doesn't work to return a float - it must be a double.
double get_time_sec()
{
    struct timeval t;
    struct timezone tzp;
    gettimeofday(&t, &tzp);
    return t.tv_sec + t.tv_usec*1e-6;
} // end get_time_sec
```

- The output of fibonacci.c demonstrated that the recursive implementation (FibonacciR) is more expensive than the loop implementation (Fibonacci), especially when the recursion is deep (after entering 50).



Stack Overflow

- Stack overflow happens when a program hits its stack space limit.
- When an interpreted language hits the stack space limit the interpreter generally throws an exception and terminates.
- Compiled languages depend upon the operating system to determine when a stack overflow has occurred.
- Older operating systems without adequate memory management would often simply fail as the stack grew over important code, causing an unrecoverable system-wide crash.
- Modern operating systems and memory systems give each program its own virtual stack space; when a program overflows its own stack of the OS can isolate the program and terminate it.
- Show task2.c. Segment fault, overflow the stack.

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

int main(int argc, char *argv[]) {

  int i;
  int bi  = 0x05050505;
  int ai  = 0x0A0B0C0D;
  unsigned char *ptr;


  ptr = (unsigned char*)&ptr;
  for (i=0; i<10000; i++) {
    printf( "%d: %p (%d)\n", i, (ptr+i), *(ptr+i));
  }

  return 0;
}
```

Heap Overflow

- Heap overflow happens when there is not available heap space that meets the required size.
- Heap is a consecutive space. Some areas are used, and others are available.
- **malloc** must obtain an unused section of the heap of the requested size and return the pointer to the beginning of that area.
- malloc needs to seek the available heap space that is large enough to the required memory size. If there is no large enough space, malloc returns null, and the heap overflow happens, even if there are smaller available pieces of memory in the heap.
- So, it is important to invoke free after done with the dynamically allocated memory. **free** marks the area of the heap that had been used by the given pointer as unused.

- Allocating memory takes time, especially when you allocate a large chunk of memory.
- Show memory.c example, run without comment out free(other_array), run with free(other_array) commented out.

```c
/**
 * memory.c
 * Learn about the layout of the heap.
 * Push the heap to its boundaries, and look at timing.
 */

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

int main() {
    int i;
    double t1, t2;

    // allocate space for n * subn ints
    unsigned long n = 100000;
    unsigned long subn = 10000000;
    unsigned long int other_n = n * subn;
    int **array;
    array = (int **)malloc( n * sizeof(int *) );

    t1 = get_time_sec();
    for (i = 0; i < n; i++) {
        array[i] = (int *)malloc(subn * sizeof(int) );
    }
    t2 = get_time_sec();
    printf( "Allocated %lu arrays of %lu ints in %f seconds\n",
                n, subn, t2-t1 );

    for (i=0; i < n; i++) {
              free( array[i] );
    }
    free(array);

    int num_times = 35;
    int *other_array = NULL;
    t1 = get_time_sec();
    for (i = 0; i < num_times; i++)   {
        other_array = (int *)malloc( other_n *sizeof(int) );
        if (other_array == NULL)
            printf( "Failed to allocate large array at iteration %d\n", i );
        //free(other_array);
    }
    t2 = get_time_sec();
       if (other_array != NULL)
              printf( "Allocated one array of %lu ints is %f seconds on average\n",
                   other_n, (t2-t1)/35 );

}
```

```
$ ./a.out
Allocated 100000 arrays of 10000000 ints in 0.112327 seconds
Allocated one array of 1000000000000 ints is 0.007313 seconds on average


$ ./a.out
Allocated 100000 arrays of 10000000 ints in 0.109065 seconds
a.out(30746,0x7fffc580c3c0) malloc: *** mach_vm_map(size=4000000000000) failed
(error code=3)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
Failed to allocate large array at iteration 34
```

- The output indicates that allocating memory takes time, and allocating large pieces of memory takes more time.
- It also demonstrated that without invoking free appropriately, the program may run into heap overflow.

- When heap overflow happens or while the program is looking for available space, the computer will trigger garbage collection.