Objectives:

(a) Explain the purpose of the heap and describe how memory on the heap is allocated.

I. The Heap

We mentioned a few lectures back the fact that when a program is to be executed, the OS reserves a block of main memory for the program’s use. This block of memory is then partitioned into segments.

The “text” segment holds the actual program (the machine language instructions which we can view as assembly-language instructions.) The “stack” segment is the memory that the program has available to store information during execution. For example, the program’s variables are stored on the stack.

Once a C program is compiled and the corresponding machine code generated, the amount of space for the text segment is fixed. Similarly, the compiler knows about all variables that we declare in our program, so the compiler can make room for these variables on the stack as soon as the function that uses these variables is invoked. The precise placement of data on the stack is completely controlled by the compiler.
Oftentimes when we run a program, we will want to use memory that cannot be anticipated in advance by the compiler (if the compiler had anticipated it, it would have reserved space for it on the stack). For example, suppose we want our program to use an array of characters, but the size of the array is a value that the user will enter at the keyboard while the program is running. The compiler cannot, in advance, predict the size of the array, since the size will depend on whatever value the user happens to enter.

So, we need an additional segment of memory available for the programmer to directly control. This additional segment is called the **heap**. The heap, like the stack, varies in size—it will grow and shrink based on the memory needs of the user as the program is running. The heap and the stack, in fact, grow towards each other.

Note that the total space allocated for the heap and the stack is fixed so, in a sense, the heap and the stack compete with each other for space. As items are added to the heap, the size of the heap grows and the "bottom of the heap" moves to a higher memory address. As items are added to the stack, the size of the stack grows and the "top of the stack" moves to a lower memory address.

### Practice Problem

Consider the picture above, showing program 1 in memory.

(a) How does the CPU keep track of the program's proper location within the text segment?

(b) How does the CPU keep track of where the stack is located in main memory?

**Solution:**

(a)  

(b)
The preceding question might leave you wondering: *How does the CPU keep track of where the heap is located in main memory?* The answer to this question is: It doesn't. It is up to the programmer to keep track of the heap. The point bears repeating: The compiler takes care of the stack, *YOU* (the programmer) must take care of the heap.

To allocate memory on the heap, we use the `malloc` function. We tell the `malloc` function the number and type of the space we need (e.g., “space for 6 integers” or “space for 25 characters”) and `malloc` returns a pointer to the start of the memory that is allocated on the heap for this purpose.

For example, to allocate space for 6 integers (which requires 24 bytes), we would use:

```c
int *ptr1;
ptr1 = (int *) malloc(24);
```

After these two lines of code execute, `ptr1` will hold the address to space on the heap for six integers. Note that the argument to the `malloc` function is the number of bytes we would like to allocate on the heap.

**Practice Problem**

Write a snippet of C code that will allocate space on the heap for 25 characters.

Solution:

```
int *ptr2;
ptr2 = (int *) malloc(25);
```

**Practice Problem**

Which segment of memory is physically highest (i.e., has the smallest addresses)?

(a) Heap  
(b) Stack  
(c) Text Segment  
(d) Registers  

Solution:

**Practice Problem**

In which direction does the heap grow?

(a) From the bottom (larger memory address) up (to a smaller memory address).  
(b) From the top (smaller memory address) down (to a larger memory address).  
(c) It depends on the corresponding number and types of variables currently allocated on the stack.  
(d) It depends on the prolonged effects of solar and liquescent additives combined with the chemical makeup of the heap.

Solution:
This whole notion of using the heap may seem mysterious, so let's look in gory detail at an example. Our goal is to write a program that accepts, as command line arguments, a number of bytes to allocate on the heap (to hold character data), and a text string to place in that newly allocated memory.

For example, if we executed our program (./a.out) with command line arguments as shown:

```sh
midshipman@EC310-VM:~ $ ./a.out 10 cyber2
```

the program would allocate 10 bytes on the heap, store the characters "cyber2" at this location, and output a message telling us the starting address for our 10 byte allocation (the address that will contain the c in "cyber2").

Here is the program to accomplish this. We will examine the program line-by-line.

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

int main( int argc , char *argv[ ] )
{
    char *ptr ;
    int size;
    size = atoi( argv[ 1 ] ) ;
    ptr = ( char* ) malloc( size );
    strcpy( ptr , argv[ 2 ] );
    printf("The following is stored on the heap at address %x:", ptr);
    printf( "%s \n\n" , ptr );
    free( ptr ) ;
}
```

**Practice Problem**

When we run the program above by entering:

```sh
./a.out 10 cyber2
```

(a) What is the value of argc?
(b) What are the values of argv[ 1 ] and argv[ 2 ]?
(c) What are the types of argv[ 1 ] and argv[ 2 ]?

Solution: (a)
(b)
(c)
On line 6 we declare a pointer to a character named `ptr`. Our intent (as we will soon see) is that `ptr` will point to the first character in a string of characters.

On line 8 we declare an integer named `size`, and then on line 10 we set `size` to be equal to the integer value of `argv[1]`. Recall that the intent of the second command line argument, `argv[1]`, is to specify the number of bytes that we wish to reserve on the heap. We want to reserve 10 bytes on the heap, so we typed in 10 as the second command line argument. It is important to remember, though, that all command line arguments are stored as strings. So we have to convert `argv[1]` to an integer. You can convert a string to an integer using the function `atoi` (which stands for ASCII to integer).

Line 12 then reserves 10 bytes on the heap, and stores the starting address of the first byte in `ptr`. At this point, the program's memory is as follows:

![Diagram showing memory layout](image1)

Line 14 then copies the string `argv[2]` into the memory starting at `ptr`. After line 14, the program's memory looks as follows:

![Diagram showing memory layout](image2)
On lines 16 and 18 we print out the address of `argv[2]` on the heap. Here is the output:

```
midshipman@EC310:~/work $ ./a.out 10 cyber2
The following is stored on the heap at address 804a008: cyber2
```

So we can refine the picture of this program in memory:

There is one last (important!) point about the heap. If our program no longer needs memory that was allocated on the heap, it should free it up so that it can be reused. This is done with the `free` function. For example, to free the heap memory in the prior program, we would use the line of code:

```c
free( ptr ) ;
```

So, our final program includes line 20.

**Practice Problem**

Suppose we run the program shown above with the debugger, and set a breakpoint at line 16. Which of the following is a possible value stored in the instruction pointer `eip`?

(a) 0x0804848c  (b) 0xbffff810  (c) 0x0804a010

Solution:

**Practice Problem**

Suppose we run the program shown above with the debugger, and set a breakpoint at line 16. Which of the following is a possible address for where the variable `size` is stored?

(a) 0x0804848c  (b) 0xbffff810  (c) 0x0804a010

Solution:
Practice Problem

The above picture of the stack shows that the variable size is stored "above" (i.e., at lower memory) than ptr. How do we know that this must be the case?

Solution:

For today's lab, we have to add a little bit to your C repertoire. For today, we have to cover the string compare command and the syntax for passing an array as an argument to a function. It’ll be fun.

II. More Fun with C

1. A new string command  Recall earlier that we were able to enter or change the value of strings using the strcpy command. Specifically, the command

   strcpy( s1, s2 );

   copies the string s2 to the string s1. Another useful command is the string compare command, strcmp. The command

   value = strcmp( s1, s2 );

   Compares the strings s1 and s2 character by character. The function returns an integer greater than zero if s1 > s2 and returns an integer less than zero if s1 < s2. Perhaps most importantly, the function returns zero if the two strings are equal (i.e., identical). To use these functions, you must have the preprocessor directive:

   #include <string.h>

Practice Problem

What is the output of the following program?

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

int main(int argc, char *argv[])
{
    char string1[ ] = "Happy" ;
    char string2[ ] = "Joyous" ;
    char string3[ ] = "Happy Times" ;
    char string4[ ] = "Happy" ;

    if(strcmp( string1 , string2 ) == 0)
        printf("\n String 1 and String 2 match\n");
    else
        printf("\nString 1 and String 2 do NOT match\n");

    if(strcmp( string1 , string3 ) == 0)
        printf("\n String 1 and String 3 match\n");
    else
        printf("\nString 1 and String 3 do NOT match\n");

    if(strcmp( string1 , string4 ) == 0)
        printf("\n String 1 and String 4 match\n");
    else
        printf("\nString 1 and String 4 do NOT match\n");
}
```
2. **Passing an array to a function**  To pass an array to a function we use the array name as an argument. In the function header, though, the type must be a pointer, since an array name is an address.

For example, suppose we had a string (an array of characters) declared as

```c
char name[10];
```

and we wanted to pass this array (as the only argument) to a `void` function named `fun`. Then the function call would be

```c
fun( name );
```

and the first line of the function definition would be

```c
void fun( char *input )
```

Thus, the argument (`name`) is passed to a parameter (`input`) which is a pointer to an array of characters.