

Slide 1 features a large blue rectangle on the right side containing the title "Heap Memory" in white. To the left of this rectangle is a vertical grey bar. Below the blue rectangle is a dark grey horizontal bar containing the text "CS2263 – Systems Software Development" in white. A small speaker icon is located in the bottom right corner of the slide.

Heap Memory

CS2263 – Systems Software Development

1



Slide 2 features a large blue rectangle on the left side containing the title "References" in white. To the right of this rectangle is a vertical grey bar. The text "Lu, Yung-Hsiang. 2015. CRC Press. New York. Pp 9-27 (Chapter 8)" is positioned to the right of the blue rectangle. A small speaker icon is located in the bottom right corner of the slide.

References

Lu, Yung-Hsiang. 2015. CRC Press. New York. Pp 9-27 (Chapter 8)

2

Lecture Learning Outcomes

At the conclusion of this presentation students should be able to:

- Explain how memory is allocated within the heap
- List the two functions that enable programmers to manage heap memory
- Write simple programs that use heap memory.



3

Stack and Heap

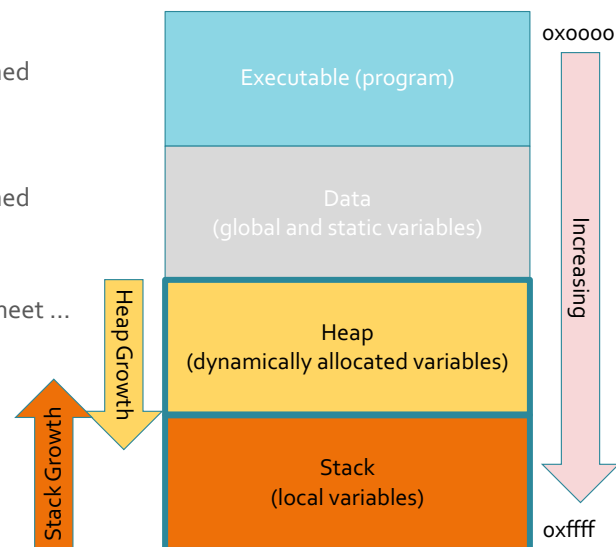
Stack

- Bottom is defined
- Grows up

Heap

- Bottom is defined
- Grows down

And when they meet ...



4

The Heap

- Stack memory is automatically allocated by the compiler
 - No programmer control
- Heap memory is managed by the programmer (that's you)
 - Allows memory to be dynamically allocated (run-time)
 - Programmer is responsible for management.
 - All management and use of heap is through pointers



5

C and the Heap

- Heap management functions: `<stdlib.h>`
- Follows the House Rules:
 - If you take it, put it back
- Getting/allocating heap memory (taking it)
 - `malloc()`
- Returning/deallocating heap memory (putting it back)
 - `free()`



6

Allocating Heap Memory: `malloc()`

```
void* malloc(size_t size)
```

- `size_t`
 - Specifies the number of bytes of memory to allocate
 - Is an unsigned int value, generally matching the target architecture of the compiler (i.e. 32-bit, or 64-bit)
 - is the datatype returned by `sizeof()`
- `void*`
 - The return type is an address
 - Is an untyped pointer
 - Holds the address of the start of the memory
 - Returns NULL if unsuccessful



7

Allocating Heap Memory: `free()`

```
free(void* ptr)
```

- `ptr`
 - The starting location of the previously allocated memory
- There is no return value
 - No way to check if successful
 - Don't worry, be happy.



8

Example Use I: single variable

```
// allocate space for a float on the heap,
// return a pointer to it,
// cast it to a float* ,
// store it in f;
float* pf = (float*) malloc( sizeof(float) );
if(pf == (float*)NULL){
    fprintf(stderr, "Memory allocation failed. Program terminating.");
    return EXIT_FAILURE;
}
printf("Please enter a float value: ");
scanf("%f",pf); // pf is a pointer, with space allocated to it
printf("%f it is!\n", *pf);
free(pf); // all done with fp
```



9

Example Use II: an array

```
// allocate space for an integer [] that hold MAXVALS ints on the heap,
// return a pointer to it, cast the void* to a int* , and store it in arr;
int iErr;
int* arr = (int*) malloc( sizeof(int)*MAXVALS );
if(pf == (float*)NULL){
    fprintf(stderr, "Memory allocation failed. Program terminating.");
    return EXIT_FAILURE;
}
printf("Please enter %d integer values: ", MAXVALS);
for(int i=0; i<MAXVALS; i++){
    iErr = scanf("%d",&arr[i]); // arr is a pointer, with space allocated
    if(iErr != 1){
        fprintf(stderr, "Read failed. Program terminating.");
        free(arr);
        return EXIT_FAILURE;
    }
    printf("%d it is!\n", arr[i]);
}
free(arr); //all done with arr
```



10

Example Use II: a string

```
// allocate space for an char [] of that hold MAXVALS chars on the heap
// (with room for the NULL), return a pointer to it, cast it to a char* ,
// and store it in s.
// Note that although s isn't YET a string, that's the intention.
char* s = (char*) malloc( sizeof(char)*(MAXVALS+1) );
if(pf == (float*)NULL){
    fprintf(stderr, "Memory allocation failed. Program terminating.");
    return EXIT_FAILURE;
}
printf("Enter up to four characters: ");
getchar(); // flushes newline feed from last response
char c = getchar();
int i = 0;
while(c != '\n' && i < MAXVALS){ // leave room for the terminating NULL!
    s[i] = c;
    i++;
    c = getchar();
}
s[i] = (char) NULL; // terminate the string
printf("%s it is!\n", s);
free(s); // all done with s
```



11

Example Use: Array of Strings I

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

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

    char** stringList;

    // allocate an array of pointers to strings (each string is a char*)
    stringList = (char**) malloc(sizeof(char*) * argc);
    if(stringList == (char**)NULL){
        fprintf(stderr, "Memory failure, terminating");
        return EXIT_FAILURE;
    }
}
```



12

Example Use: Array of Strings II

```
for(int i=0; i<argc; i++){
    // allocate the string
    stringList[i] = (char*) malloc( sizeof(char)* (strlen(argv[i]) +1) );
    if(stringList[i] == (char*)NULL){
        fprintf(stderr, "Memory failure, terminating");
        for(int j=0; j<i; j++){
            free(stringList[j]);
        }
        free(stringList);
        return EXIT_FAILURE;
    }
    // copy commandline argument
    strcpy(stringList[i], argv[i]);
}
```



13

Example Use: Array of Strings III

```
// Did we do it? Can we print them out?
for(int i=0; i<argc; i++){
    printf("%s\n", stringList[i]);
}
```



14

Example Use: Array of Strings IV

```
// Cleaning up after ourselves, from the end to the beginning
for(int i=0; i<argc; i++){
    free(stringList[i]);
}
free(stringList);
```

