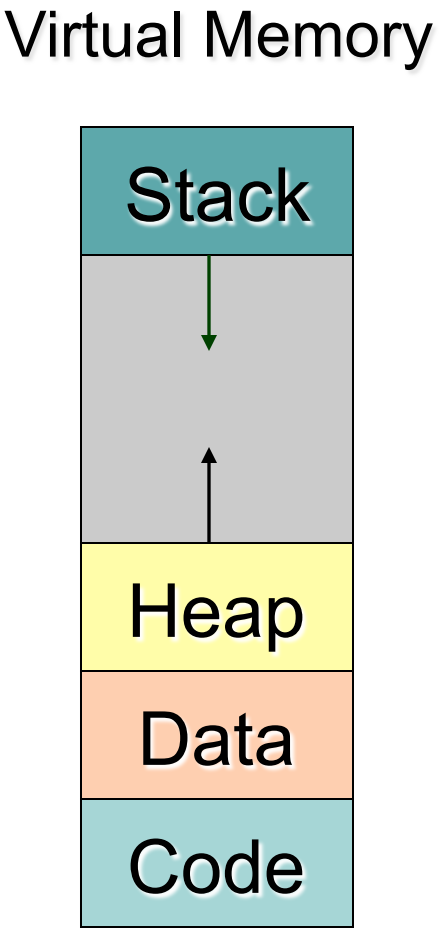
CS240: Programming in C

Lecture 8: Dynamic memory allocation.

PURDUE UNIVERSITY

Memory layout for a process

- The operating system creates a process by assigning memory and other resources
- **Stack**: keeps track of the point to which each active subroutine should return control when it finishes executing; stores variables that are local to functions
- **Heap: dynamic memory for variables that are created with *malloc, calloc, realloc* and disposed of with *free***
- **Data**: initialized variables including global and static variables, un-initialized variables
- **Code**: the program instructions to be executed



Variables

- Global – accessible in all functions, they get in ‘data memory’
- Local – declared within functions, they get allocated on the stack, they ‘disappear’ when the function returns
- Dynamically allocated – they get allocated on the heap, the user allocates and de-allocates them

Dynamic memory management

```
#include <stdlib.h>

void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

Allocate and free dynamic memory

Operations with memory

```
void *memset(void *s, int c, size_t n);
void *memcpy(void *s1, const void *s2,
  size_t n);
```

Initializing and copying blocks of memory.

void *malloc(size_t size);

- Allocates **size** bytes and returns a pointer to the allocated memory. The memory is not cleared. (Use memset to zero it.)
- The returned value is a pointer to the allocated memory, suitable for any kind of variable, or NULL if the request fails. You have to cast the pointer.

```
p = (char*) malloc(10); /* allocated 10 bytes */
if(p == NULL) {
   ….
}
```

MALLOC CAN FAIL, YOU SHOULD CHECK THAT THE RETURNED POINTER IS NOT NULL

`void *calloc(size_t nmemb, size_t size);`

- Allocates memory for an array of **`nmemb`** elements of **`size`** bytes each and returns a pointer to the allocated memory. The memory is set to zero.
- The value returned is a pointer to the allocated memory, which is suitable for any kind of variable, or NULL if the request fails.

```
p = (char*) calloc(10,1); /* allocates 10 bytes */
if(p == NULL) {
  ….
}
```

CALLOC CAN FAIL, YOU SHOULD CHECK THAT THE RETURNED POINTER IS NOT NULL

void free(void *ptr);

- Frees the memory space pointed to by **ptr**, which must have been allocated with a previous call to malloc, calloc or realloc.
- If memory was not allocated before, or if free(ptr) has already been called before, undefined behavior occurs. If ptr is NULL, no operation is performed.
- free returns no value.

```
char *mess = NULL;
mess = (char*) malloc(100);
….
free(mess);
```

FREE DOES NOT SET THE POINTER TO NULL

`void *realloc(void *ptr, size_t size);`

- Changes the size of the memory block pointed to by **`ptr`** to **`size`** bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. Unless ptr is NULL, it must have been returned by an earlier call to malloc, calloc or realloc.
- If ptr is NULL, equivalent to malloc(size);
- If size is equal to zero, equivalent to free(ptr).
- Returns a pointer to the newly allocated memory, which is suitable for any kind of variable and may be different from **`ptr`**, or returns NULL if the request fails or if size was equal to 0.
- If fails the original block is left untouched, i.e. it is not freed or moved.

`void *memcpy(void *dest,const void *src,size_t n);`

- Copies **`n`** bytes from memory area **`src`** to memory area **`dest`**. It returns **`dest`**.
- The function operates as efficiently as possible on memory areas. It does not check for overflow of any receiving memory area.

```
char buf[100];
char src[20] = “Hi there!”;
int type = 9;

memcpy(buf, &type, sizeof(int)); /* copy an int */

memcpy(buf+sizeof(int), src, 10); /* now copy 10 chars */
```

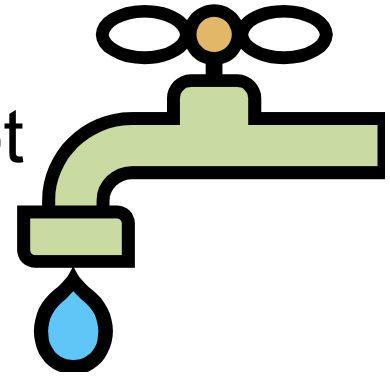
`void *memset(void *s, int c, size_t n);`

- Sets the first **`n`** bytes in memory area **`s`** to the value of **`c`** (converted to an unsigned char). It returns **`s`**.
- Operates as efficiently as possible on memory areas. It does not check for overflow of any receiving memory area.

```
memset(message, 0, 100);
```

Memory Allocation Problems

1. Memory leaks



- Dynamically allocated memory is not freed appropriately.
- If your program runs a long time (service), it will ‘eat’ memory, so it will slow down the system.

- ALWAYS WRITE THE FREE WHEN YOU WRITE THE MALLOC, decide later where the `free’ call goes.

Memory Allocation Problems

2. Deallocation bugs

- Deallocating something twice.
- Deallocating something that was not allocated, remember that if a pointer is not NULL, free will try to free the memory

- Both can cause unexpected behavior. For example, the next call to malloc will fail (!! The next malloc can be in a program while deallocating something that was not allocated can be in a library!!).

Memory Allocation Problems

3. Memory overrun

- Write in memory that was not allocated. The program will exit with segmentation fault.
- Overwrite memory: unexpected behavior.

Dynamic memory: Checklist

- Set your pointer to NULL when you declare it
- Verify that malloc succeeded
- Initialize the allocated memory
- Write the free when you write malloc
- Set pointer to null after you freed it

Readings and exercises for this lecture

K&R Chapter 5.10 for command line arguments