C Chapter 10:
Dynamic Memory Allocation

CECS130
Introduction to Programming Languages
Dr. Roman V. Yampolskiy

Dynamic memory allocation

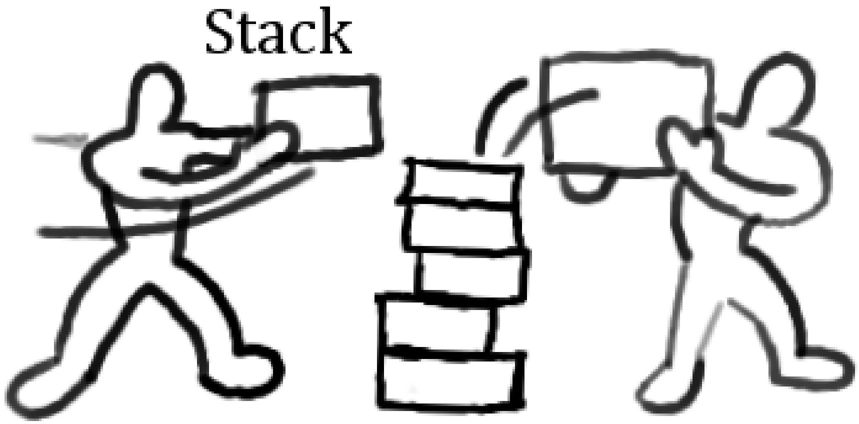
- Dynamic memory allocation is the allocation of memory storage for use in a computer program during the runtime of that program
- A way of distributing ownership of limited memory resources among many pieces of data and code.
- RAM–Random Access Memory
- Virtual Memory – Hard disk space used to simulate additional RAM
- VM provides OS with inexpensive solution for dynamic memory demands.

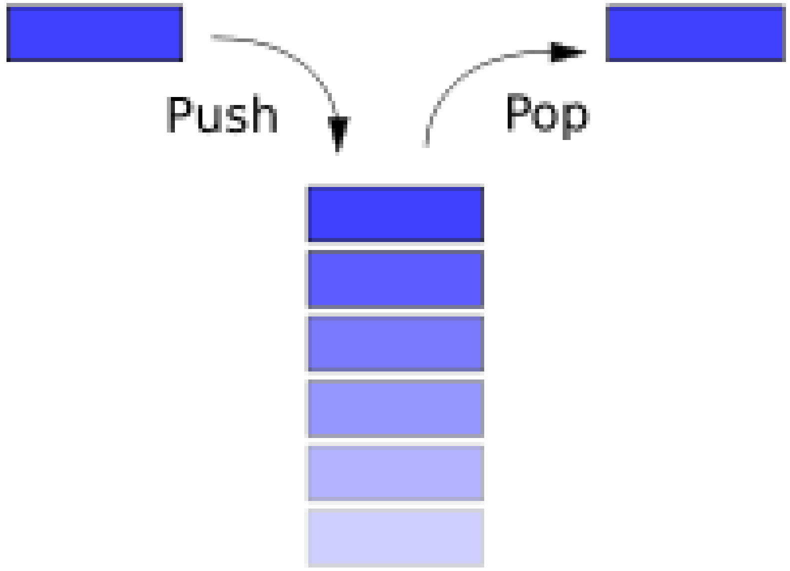
Stack

- RAM and VM provide each program with memory to be used by the program, known as the **stack**
- Stack is a dynamic grouping of memory that grows and shrinks to accommodate needs of individual programs
- Every time a function is called in the program the function's variables and parameters are pushed onto the stack
- The variables and parameters are pushed off the stack as needed for the use in the function

Stack

- Stack is a Last In First Out (LIFO) data structure





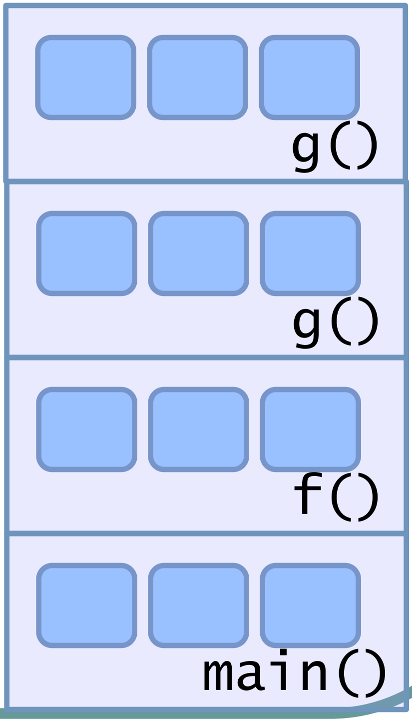
Heap

- Unallocated memory is known as Heap
- Heap is managed by the Operating System
- After your program terminates OS re-uses memory used by your program by returning it to the heap
- Dynamic memory allocation is all about retrieving and returning memory to and from the heap

Overview of memory management

- Stack-allocated memory
  - When a function is called, memory is allocated for all of its parameters and local variables.
  - Each active function call has memory on the stack (with the current function call on top)
  - When a function call terminates,
    the memory is de-allocated (“freed up”)

- Ex: `main()` calls `f()`,
  `f()` calls `g()`
  `g()` recursively calls `g()`



Overview of memory management

- Heap-allocated memory
  - This is used for ***persistent*** data, that must survive beyond the lifetime of a function call
    - global variables
    - dynamically allocated memory – C statements can create new heap data
  - Heap memory is allocated in a more complex way than stack memory
  - Like stack-allocated memory, the underlying system determines where to get more memory – the programmer doesn’t have to search for free memory space

Dynamic memory allocation

- Fixed-sized objects, where size is known at compile-time, are stored on the *stack*
- Sometimes you don't know the size you'll need for an array at compile-time
- You can request memory dynamically, at run time, from the *heap*
- Dynamic allocation can also be used to create memory for a single object (int, structure, etc.)

sizeof()

- sizeof function returns the size of a variable or data type
- Returns the number of bytes necessary to store data in memory
- sizeof can be used to find the size of any data type, variable or structure.
- Examples:

```
int temp = sizeof(int)

int myNumber = 8;
int temp = sizeof(myNumber);
```

sizeof- basic types

```
#include <stdio.h>
main() {
        int array[10];
        printf("Size of array %d bytes \n", sizeof(array));
        printf("Number of elements in array: ");
        printf("%d\n",sizeof(array)/sizeof(int));
}
```

Output: Size of array: 40 bytes

Number of elements in array: 10

sizeof- basic types

sizeof (char) = 1
sizeof (double) = 8
sizeof (float) = 4
sizeof (int) = 4
sizeof (long) = 4
sizeof (long long) = 8
sizeof (short) = 2
sizeof (void *) = 4

Note: `void *` denotes a generic pointer type

malloc()

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

- Allocate a block of `size` bytes, return a pointer to the block (`NULL` if unable to allocate block)

Checking for successful allocation

- Call to `malloc` might fail to allocate memory, if there's not enough available
- Always check!

```
#include <stdio.h>
#include <stdlib.h>
main() {
    char *name;
    name = (char *) malloc(80*sizeof(char));
    if (name == null) printf("\nOut of Memory\n");
    else printf("\nMemory allocated\n");
}
```

calloc()

```
void *calloc(size_t num_elements, size_t
  element_size);
```

- Allocate a block of `num_elements * element_size` bytes,

  initialize every byte to zero,

  return pointer to the block

  (`NULL` if unable to allocate block)

calloc(): Example

```
#include <stdio.h>
#include <stdlib.h>
// required for the malloc, calloc and free functions

int main() {
  float *calloc1, *calloc2, *malloc1, *malloc2;
  int i;

  calloc1 = calloc(3, sizeof(float));
  // might need to cast

  calloc2 = calloc(3, sizeof(float));
  malloc1 = malloc(3 * sizeof(float));
  malloc2 = malloc(3 * sizeof(float));
```

```
if(calloc1!=NULL && calloc2!=NULL && malloc1!=NULL &&
malloc2!=NULL) {

    for(i=0 ; i<3 ; i++) {
      printf("calloc1[%d] holds %05.5f, ", i, calloc1[i]);
      printf("malloc1[%d] holds %05.5f\n", i, malloc1[i]);
      printf("calloc2[%d] holds %05.5f, ", i, *(calloc2+i));
      printf("malloc2[%d] holds %05.5f\n", i, *(malloc2+i));
    }

    free(calloc1);
    free(calloc2);
    free(malloc1);
    free(malloc2);

    return 0;
  }
  else {
    printf("Not enough memory\n");
    return 1;
  }
}
```

calloc1 and calloc2 arrays should hold zeros. Contents of the malloc1 and malloc2 arrays will vary

realloc()

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

- Given a previously allocated block starting at `ptr`,
  - change the block size to `new_size`,
  - return pointer to resized block
    - If block size is increased, contents of old block may be copied to a completely different region
    - In this case, the pointer returned will be different from the `ptr` argument, and `ptr` will no longer point to a valid memory region
- If `ptr` is `NULL`, `realloc` is identical to `malloc`

- Note: may need to cast return value of `malloc/calloc/realloc`:

```
char *p = (char *) malloc(SIZE);
```

realloc()- possible outcomes

- Successful without move-
  Same pointer returned
- Successful with move-
  New pointer returned
- Not successful –
  NULL pointer returned

realloc()-Example

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

int main() {
  int *ptr;
  int i;

  ptr = calloc(5, sizeof(int));

  if(ptr!=NULL) {
    *ptr = 1;
    *(ptr+1) = 2;
    ptr[2] = 4;
    ptr[3] = 8;
    ptr[4] = 16;
    /* ptr[5] = 32; wouldn't assign anything */

    ptr = realloc(ptr, 7*sizeof(int));
```

```
if(ptr!=NULL) {
      printf("Now allocating more memory... \n");
      ptr[5] = 32; /* now it's legal! */
      ptr[6] = 64;

      for(i=0 ; i<7 ; i++) {
        printf("ptr[%d] holds %d\n", i, ptr[i]);
      }
      realloc(ptr,0);              // same as free(ptr);
      return 0;
    }
    else {
      printf("Not enough memory - realloc failed.\n");
      return 1;
    }
  }
  else {
    printf("Not enough memory - calloc failed.\n");
    return 1;
  }
}
```

realloc(): Example-Output

- **Now allocating more memory...**
  ptr[0] holds 1
  ptr[1] holds 2
  ptr[2] holds 4
  ptr[3] holds 8
  ptr[4] holds 16
  ptr[5] holds 32
  ptr[6] holds 64

free()

```
void free(void *pointer);
```

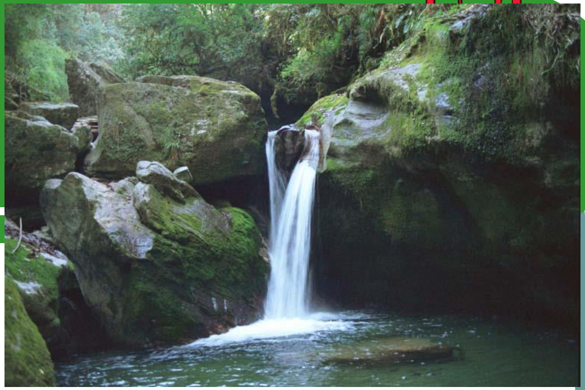
- Given a pointer to previously allocated memory,
  - put the region back in the heap of unallocated memory

- Note: easy to forget to free memory when no longer needed...
  - especially if you're used to a language with "garbage collection" like Java
  - This is the source of the notorious "memory leak" problem
  - Difficult to trace – the program will run fine for some time, until suddenly there is no more memory!

Dynamic allocation functions

- Dynamic allocation functions:
  - **`malloc`** – allocates space that is uninitialized
  - **`calloc`** – allocates spaces that is initialized with 0's
  - **`realloc`** – re-allocates space
  - **`free`** – de-allocates space
- Every **`malloc`**, **`calloc`**, **`realloc`** should have a matching call to **`free`**
- Otherwise, you have a *memory leak*

Memory leaks



- *Memory leaks* occur when you forget to call `free`
- C has no automatic garbage collection
- Particularly fatal to long-running processes that do many allocations (e.g. servers)
- Usually the result of
  - Simple forgetfulness
  - Multiple return paths
  - Reassigning the pointer without calling `free` first, esp. for in/out parameters
  - When freeing a structure, forgetting to also free the structure members
  - Not realizing when a function allocates memory that the caller is responsible for freeing

Memory segments

- Individual memory segments acquired by malloc() can be treated like array cells

```
int *numbers;
int x;
numbers = (int *) malloc(5*sizeof(int));
numbers[0] = 100;
numbers[1] = 200;
numbers[2] = 300;
numbers[3] = 400;
numbers[4] = 500;
```

Memory errors

- Using memory that you have not initialized
- Using memory that you do not own
- Using more memory than you have allocated
- Using faulty heap memory management

Using memory that you have not initialized

- Uninitialized memory read
- Uninitialized memory copy
  - not necessarily critical – unless a memory read follows

```
void foo(int *pi) {
  int j;
  *pi = j;
  /* UMC: j is uninitialized, copied into *pi */
}

void bar() {
  int i=10;
  foo(&i);
  printf("i = %d\n", i);
  /* UMR: Using i, which is now junk value */
}
```

Using memory that you don’t own

- Null pointer read/write

```
typedef struct node {
  struct node* next;
  int val;
} Node;

int findLastNodeValue(Node* head) {
  while (head->next != NULL) { /* Expect NPR */
      head = head->next;
  }
  return head->val;
}
```

What if head is NULL?

Using memory that you haven’t allocated

- Array bound read/write

```
void genABRandABW() {
  const char *name = “Safety Critical";
  char *str = (char*) malloc(10);
  strncpy(str, name, 10);
  str[11] = '\0'; /* Expect ABW */
  printf("%s\n", str); /* Expect ABR */
}

Output: Safety cri\
```

Faulty heap management

- Freeing non-heap memory
- Freeing unallocated memory

```
void genFNH() {
  int fnh = 0;
  free(&fnh); /* Expect FNH: freeing stack memory */
}
void genFUM() {
  int *fum = (int *) malloc(4 * sizeof(int));
  free(fum+1); /* Expect FUM: fum+1 points to middle of a
  block */
  free(fum);
  free(fum); /* Expect FUM: freeing already freed memory
  */
}
```

malloc: example-strdup

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

char* strdup(const char* s) {                    //duplicates a string
        char* p = 0;
        p = (char*) malloc(strlen(s)+1);
        if (p) strcpy(p, s);
        return p;
}
void main() {
        /* Make a copy: strdup allocates memory! */
        char* stringCopy = strdup("Three");
        printf("One Two %s\n", stringCopy);
        /* Deallocate memory */
        free(stringCopy);
        stringCopy = NULL; /* So we don't accidentally use it */
        system("pause");
}
```

Why +1? Why check p?

The End!