KU LEUVEN

C: an introduction

Memory Management

Dynamic Memory

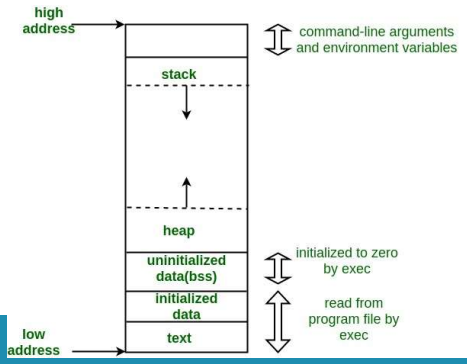
- Often a program cannot know how much memory it will need in advance.
- Defining oversized buffers is one solution,

```
char buffer[SIZE];
```

  but may not always be a feasible solution.
- Dynamic memory allocation provides memory on demand at runtime. Gives programmer complete control over object lifetime.

Different Memory Areas

- See also:
  - https://www.gribblelab.org/CBootCamp/7_Memory_Stack_vs_Heap.html
  - https://www.geeksforgeeks.org/memory-layout-of-c-program/



Dynamic Memory Functions

- The standard library (`stlib.h`) provides a number of functions for using dynamic memory.
- The `malloc` function takes a size in bytes and returns a pointer to a block of free memory of that size. This dynamically allocated memory is uninitialized and can only be accessed through pointers.
- Syntax: `ptr = (castType*) malloc(size);`

```
int * p = malloc(10 * sizeof(int));

int * ptr;
ptr = (int*) malloc(100 * sizeof(int));
```

malloc()

- If **`malloc()`** succeeds, it returns a pointer to the allocated memory block.
- If it fails, it returns **`NULL`**.
- It is rare for memory allocation to fail on modern machines with virtual memory. Always check the return value.

free()

- `free()` releases memory that was allocated by `malloc()`.
- Free takes in the address of the start of the memory block you allocated. (the one returned by malloc). If you input any other memory location. It will result in crash.
- Syntax:

```
void free(void *p);


int *p = (int *)malloc(10 * sizeof(int));
...
free(p);
```

```c
#include <stdio.h>
#include <stdlib.h>
// demo_malloc_1.c

int main(){
  int *p;
  /* Allocate 4 bytes */
  p = (int *) malloc( sizeof(int) );
  if (p == 0)
  {
    printf("ERROR: Out of memory\n");
    return 1;
  }
  printf("enter an integer");
  scanf("%d", p);
  printf("content = %d    address = %p \n", *p, p);
  /* This returns the memory to the system.*/
  free(p);
}
```

```
frankvp@CRD-L-08004:.../memory$ gcc demo_malloc_1.c -o demo_malloc_1
frankvp@CRD-L-08004:.../memory$ ./demo_malloc_1
enter an integer2
content = 2    address = 0x55daa81602a0
frankvp@CRD-L-08004:.../memory$ ./demo_malloc_1
enter an integer2.3
content = 2    address = 0x555fe0dd22a0
frankvp@CRD-L-08004:.../memory$ ./demo_malloc_1
enter an integer890800323
content = 890800323    address = 0x55ef79dab2a0
frankvp@CRD-L-08004:.../memory$
```

KU LEUVEN

```c
/*
demo_malloc_4.c
http://anee.me/dynamic-memory-allocation-in-c/
*/

#include <stdio.h>
#include <stdlib.h> // for access to the memory allocation functions

#define NO_OF_CHARS 20

int main()
{
 void * upper_bound; // To avoid invalid reads.
 char * char_ptr;

 // Tip 1: Always cast the returned pointer from malloc, Most of the compilers are kind enough to give you warnings for that.
 // Tip 2: Always check the output of malloc
 char_ptr = (char *)malloc(sizeof(char) * (NO_OF_CHARS + 1)); // we need an extra byte for the NULL, same as "char array[NO_OF_CHARS]"

 if (char_ptr == NULL) {
    printf("Failed to allocate memory for chars.");
 }

 // char_ptr
 upper_bound = char_ptr + NO_OF_CHARS;

for (; char_ptr < (char *)upper_bound; char_ptr++) {
    *char_ptr = 'a';
 }
 *char_ptr = '\0'; // terminate the string
 printf("This should display a string of %d A's : %s\n", NO_OF_CHARS, char_ptr - NO_OF_CHARS);

// Tip 3: Always free your pointers
 free(char_ptr - NO_OF_CHARS);

 // Note: Free takes in the address of the start of the memory block you allocated. (the one returned by malloc)
 // if you input any other memory location. It will result in crash.

 return 0;
}
```

```
frankvp@CRD-L-08004:.../memory$ gcc demo_malloc_4.c -o demo_malloc_4
frankvp@CRD-L-08004:.../memory$ ./demo_malloc_4
This should display a string of 20 A's : aaaaaaaaaaaaaaaaaaaa
frankvp@CRD-L-08004:.../memory$
```

KU LEUVEN

calloc()

- `calloc()` behaves like `malloc()` but initialises the array to zero. Stands for **c**lear **alloc**ate.
- Has a slightly different interface than `malloc()`.

```
void *calloc(size_t n, size_t size)
```

```
int *p = calloc(10, sizeof(int));
```


realloc()

- `realloc()` is used to adjust the size of a memory block previously allocated by `malloc()`, `calloc()`, or `realloc()` itself.

```
void *realloc(void *p, size_t size)
```

- A versatile function.
  - If `p` is NULL. Acts like `malloc()`.
  - If reallocation fails, returns `NULL`, and preserves original memory block.
  - If `size` is 0. Acts like `free()` (and returns `NULL`).

Dynamic Memory Management

- Managing dynamic memory is entirely the responsibility of the programmer.
- Manage:
  - Pointers to each block.
  - Allocation and deallocation (object lifetimes).
  - Array length records.
- Without care, dynamic memory can result in many bugs. Usually very hard to track down.

Memory Errors

- Derefencing a pointer to an invalid address.
  - Change the value of some arbitrary memory location will have an arbitrary effect. Known as “memory corruption”.
- Dereferencing a pointer that has been freed.
  - For example, pointers invalidated by `realloc()`.
- Dereferencing a NULL pointer.
  - Will cause an immediate crash on most systems.
- Freeing a pointer that has already been freed.
- Freeing a pointer to memory that is not dynamic (e.g., stack, constant data).
- Neglecting to free dynamic memory (memory leak).
- Out-of-bounds array accesses.
  - Off-by-one indexing errors are a common problem. The result of an incorrect indexing algorithm.

Good Practice

- Every malloc() has an associated free().
  - Avoid memory corruption and memory leaks.
  - Best to have calls to malloc() and free() in the same function rather than expecting the calling function to free memory.
  - Often good to define a create() function and a companion destroy() function for complex objects.
- Pointers should be initialised when they are defined.
  - Initialise to a valid address or to NULL.
  - NULL explicitly marks a pointer as “points to nowhere”.
- Pointers should be assigned to NULL after being freed.
  - Prevents multiple free problem. `free(NULL)` has no effect.