

C Memory Management

Stack and Heap, Memory Management Functions, Memory Leaks, Using Valgrind



SoftUni Team
Technical Trainers
Software University
<http://softuni.bg>



Table of Contents

1. Process Memory – Stack and Heap
2. Memory Management Functions
 - `malloc()`, `calloc()`, `realloc()`, `free()`, `memset()`, etc.
3. Memory Leak, Buffer Overflow, Dangling Pointer
4. Memory Profilers
 - Valgrind



```
int is_prime(int num)
{
    int top = sqrt(num);
    for (int i = 2; i <= top; i++)
        if ((num % i) == 0)
            return 0;

    return 1;
}

void print_all_primes(int start, int end)
{
    for (int i = 5; i <= 10; i++)
        if (is_prime(i))
            printf("%d ", i);
}

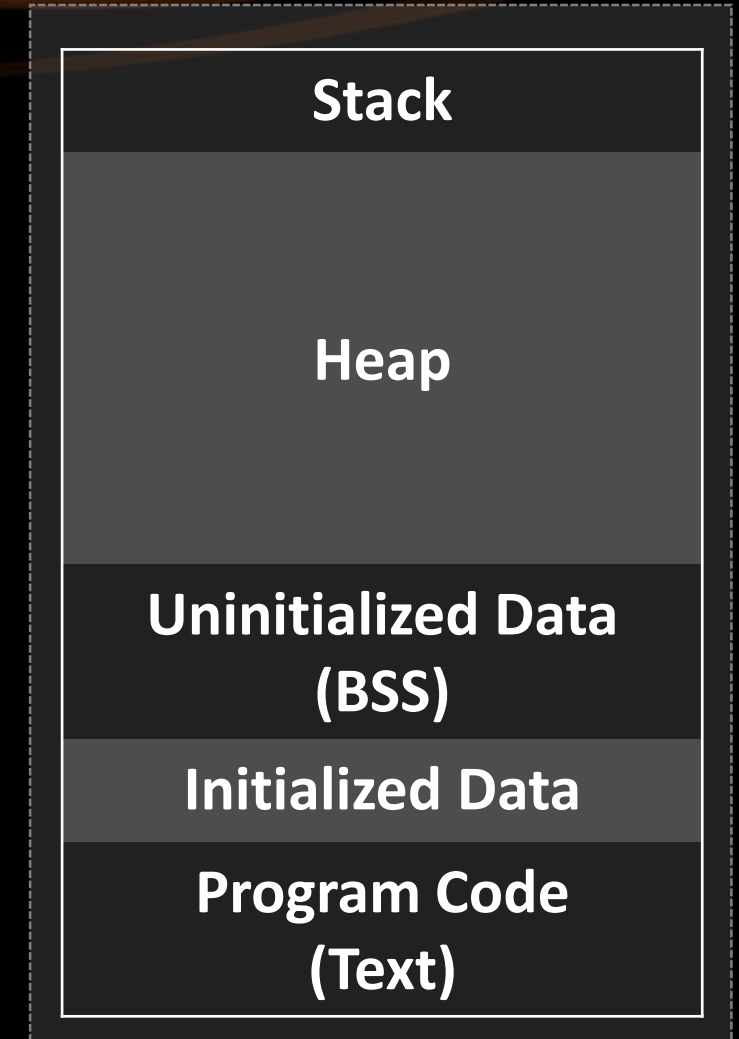
int main()
{
    print_all_primes(5, 10);
    return 0;
}
```

23	00	85	255
00	13	33	00
144	75	54	133
00	21	65	00
00	00	00	01

Process Memory

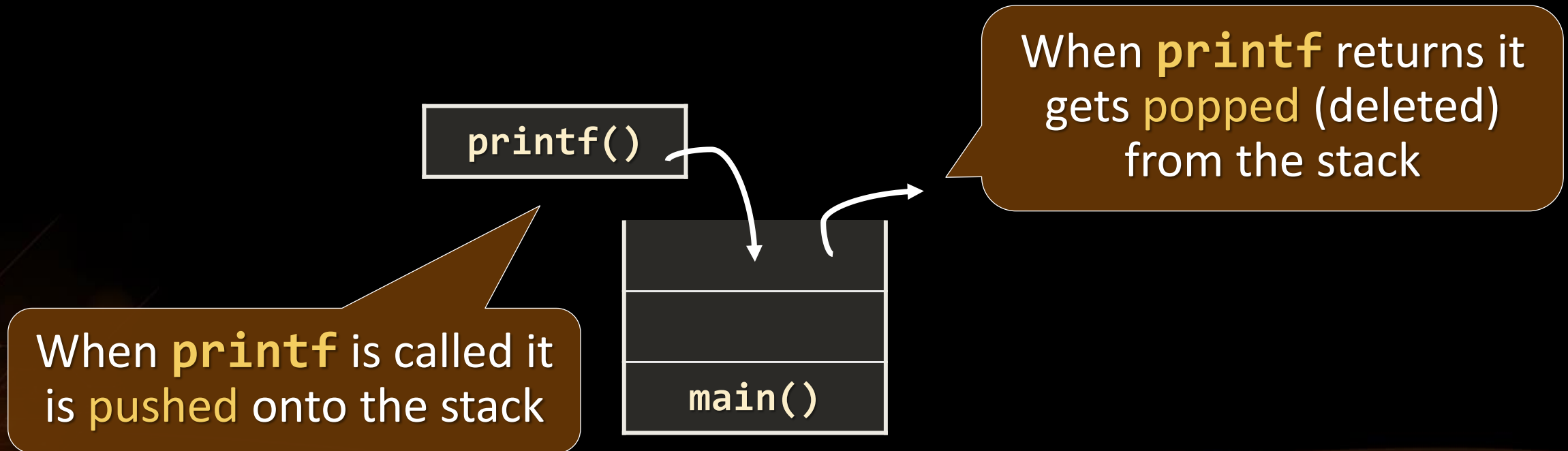
Process Memory

- All Linux processes are granted a memory space to execute on, comprised of:
 1. Stack – holds called functions and their local variables
 2. Heap – holds dynamically allocated objects
 3. Uninitialized Data – uninitialized global variables
 4. Initialized Data – initialized global variables
 5. Program Code (binary)



The Stack

- The **stack** is a small **fixed-size** chunk of memory (e.g. 1MB)
 - Keeps the currently called functions in a stack data structure
 - Changes as the program enters / exits a function



The Heap

- The **heap** (dynamic memory) holds dynamically allocated objects during execution
 - Can grow by requesting more memory from the operating system
 - Suitable for keeping large blocks of memory
 - In C, the programmer has to **manually** allocate and free memory on the **heap**
 - Functions from **<stdlib.h>** header

Memory Management Functions

malloc

- **malloc(size_t size)** – allocates **size** bytes of memory and returns a **void*** pointer to them
 - E.g. allocating 10 characters on the heap:

```
char *string = malloc(sizeof(char) * 10);  
if (string != NULL)  
{  
    // Use allocated memory  
}
```

- Note: **malloc()** returns **NULL** if no memory is available
 - Always check if the allocation is successful!

malloc – Example

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

int main()
{
    int n = 5;
    int *numbers = malloc(sizeof(int) * n);
    if (numbers)
    {
        for (int i = 0; i < n; i++)
            scanf("%d", &numbers[i]);
        // ...
        free(numbers);
    }
    return 0;
}
```

Free allocated memory
once done using it

malloc – More

- **malloc()** returns a pointer to uninitialized garbage memory
 - It may contain memory from old data

```
char *str = malloc(100);  
size_t length = strlen(str);
```



- Above code has undefined behavior – ***str** may point to any data
 - Null terminator '**\0**' could be anywhere (or even missing)

```
char *str = malloc(100);  
str[0] = '\0';  
size_t length = strlen(str);
```



calloc

- **calloc(size_t num, size_t size)** – allocates **num** blocks of memory, each **size** bytes long
 - Similar to **malloc()**, but zero-initializes the allocated memory
 - Eliminates the danger of reading garbage memory
 - Returns **NULL** if allocation fails

```
float *xCoords = calloc(10, sizeof(float));  
if (xCoords)  
{  
    // ...  
}
```

realloc

- **realloc(void *ptr, size_t newSize)** – copies the memory pointed by ***ptr** to a new block of memory
 - If successful, automatically frees the old block of memory
 - Returns **NULL** if the allocation fails

```
int *nums = malloc(sizeof(int) * n);
if (nums)
{
    int *newNums = realloc(nums, sizeof(int) * n * 2);
    if (newNums)
        ...
}
```


free

- **free(void *ptr)** – marks the memory pointed to by ***ptr** as free to the operating system
 - After that, that memory should no longer be read/written (can damage data or segfault)

```
char *string = malloc(sizeof(char) * 10);
if (string)
{
    // Use allocated memory
    free(string);
}
```

Using free

- Rules when using **free()**:
 - Use only on pointers obtained with **malloc()**, **calloc()** or **realloc()**
 - Do NOT use the pointer after you have free'd it
 - Do NOT use on stack-allocated or read-only memory pointers, e.g.:

```
char *name = "Gosho";  
int nums[] = { 2, 3, 4 };  
free(name); // Segmentation fault, do not free!  
free(nums); // Segmentation fault
```



- Always free heap-allocated resources once you're done using them

free()

Live Demo

Matrix on the Heap

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

#define ROWS 4
#define COLS 4

void error(char *message)
{
    perror(message);
    exit(1);
}
```

Prints message to the
standard error stream

Terminates the program
with status code 1

(example continues)

Allocating the Matrix (2)

```
int main()
{
    int **matrix = calloc(ROWS, sizeof(int*));
    if (!matrix)
    {
        error("Error allocating matrix");
    }

    for (int i = 0; i < ROWS; i++)
    {
        matrix[i] = calloc(COLS, sizeof(int));
        if (!matrix[i])
            error("Error allocating matrix row");
    }
}
```

(example continues)

Using and Disposing the Matrix (3)

```
matrix[0][1] = 5;
matrix[3][3] = 16;

for (int r = 0; r < ROWS; r++)
{
    for (int c = 0; c < COLS; c++)
    {
        printf("%-5d", matrix[r][c]);
    }
    printf("\n");
}

for (int i = 0; i < ROWS; i++)
    free(matrix[i]);
free(matrix);

return 0;
}
```

Matrix on the Heap

Live Demo

memset

- `memset(void *ptr, int n, size_t size)` – sets `n` bytes to `size`, starting from the address of `*ptr`
- Returns a pointer to the memory area

```
int memSize = sizeof(int) * 20;  
int *array = malloc(memSize);  
if (array)  
{  
    memset(array, 0, memSize);  
    // Use memory...  
    free(array);  
}
```

`calloc()` does this internally

memcpy

- `void *memcpy(void *s1, const void *s2, size_t n)` – copies `n` characters from `*s2` to `*s1`

```
int nums[] = { 5, 6, 10, 13, 2 };  
int size = sizeof(nums);
```

```
int *copy = malloc(size);  
memcpy(copy, nums, size);
```

```
for (int i = 0; i < 5; i++)  
{  
    printf("%d\n", copy[i]);  
}
```

Other Memory Functions

- **void *memmove(void *s1, const void *s2, size_t n)** – copies **n** characters from ***s2** to ***s1**
 - The copy is performed as if the characters from ***s2** were copied into a temporary array first
- **int memcmp(const void *s1, const void *s2, size_t n)** – compares **n** bytes from ***s1** to bytes of ***s2**

Common Errors When Working with Memory

Memory Leak

- A **memory leak** occurs when the program does not release unused resources, e.g.:
 - Not closing a file or network socket after using it
 - Not freeing allocated heap memory



```
int process()
{
    char *bytes = malloc(50);
    ...
    // Does not free
    return 0;
}
```

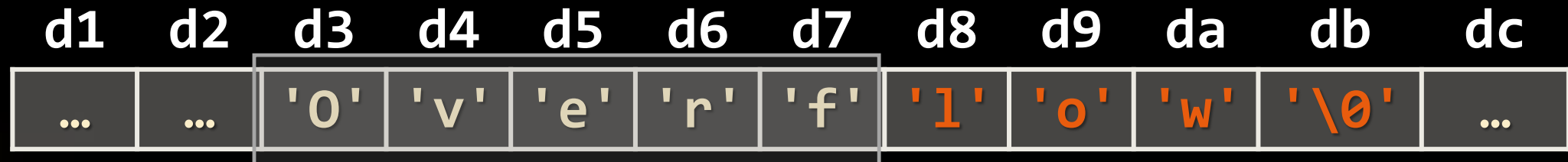


Allocated memory pointed to by ***bytes** is lost forever

Buffer Overflow

- A **buffer overflow** happens when the program overruns a buffer's boundary and writes to adjacent memory

```
char *buffer = malloc(5);  
strcpy(buffer, "Overflow");
```



***buffer**

Overwrites adjacent
memory cells

Dangling Pointer

- **Dangling pointers** are pointers that no longer point to a valid block of memory, e.g.:
 - Using a pointer after it is freed:
 - Returning a pointer to a local function variable:

```
char *text = malloc(128);  
...  
free(text);  
...  
printf("%s", text);
```



```
char *get_string()  
{  
    char str[100];  
  
    return str;  
}
```



Dangling Pointer – Example

```
#define MIN 5
#define MAX 7

int *get_range(int min, int max)
{
    int nums[max - min + 1];
    for (int i = 0; i <= max - min + 1; i++)
        nums[i] = min++;
    return nums;
}

int main()
{
    int *nums = get_range(MIN, MAX);
    for (int i = 0; i <= MAX - MIN + 1; i++)
        printf("%d ", nums[i]);
    return 0;
}
```



Return pops **get_range()** from the stack (along with **nums**)

Prints garbage memory

Dangling Pointers

Live Demo

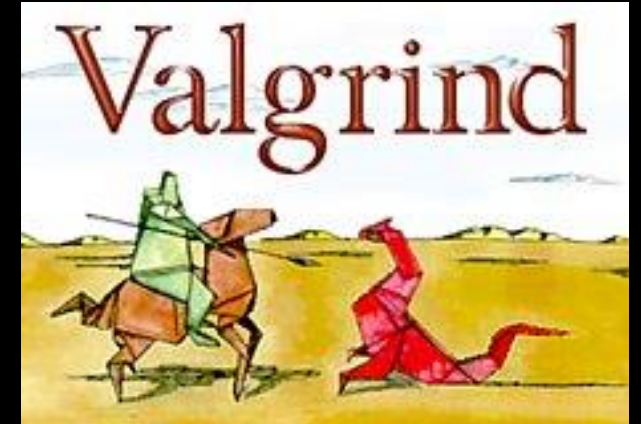


Memory Profilers

Valgrind

Memory Profilers

- **Memory profilers** analyze the program's memory allocations, reads, writes and deallocations during execution
- **Valgrind** is a multiplatform C/C++ memory profiler
 - Free and open source (GPL license)
 - Features memory checking tools for finding memory leaks, dangling pointers and buffer overflows
 - <http://www.valgrind.org/>



Using Valgrind

- Can be installed through the package manager

- *buntu: `sudo apt-get install valgrind`

- Run command: `valgrind [param1 param2 ...] ./{executable}`

- Parameters:

- **--leak-check=yes** – checks for memory leaks
- **--track-origins=yes** – tracks the source of uninitialized data
- **--log-file="log.out"** – redirects the output to a file
- **--track-fds=yes** – displays non-closed files/sockets



Using Valgrind

Live Demo

Stack Smashing

- Valgrind does not offer protection against stack smashing
- GCC adds protection when run when run with the **-fstack-protector-all** flag

```
User user;  
user.isAdmin = 0;  
gets(user.name);  
if (user.isAdmin)  
{  
    // .. Do sensitive stuff  
}
```

```
typedef struct User {  
    char name[10];  
    int isAdmin;  
} User;
```



- Note: NEVER use **gets()**!

Stack Smashing

Live Demo

Summary

- **malloc()**, **calloc()** and **realloc()** allocate a memory block on the **heap** and return a pointer to it
- **free()** releases the occupied memory block
- Should be used only on pointers obtained by the above functions
- **Valgrind** is a memory profiling tool that tracks down errors
 - **Buffer overflows** – writing outside the boundaries of a buffer
 - **Memory leaks** – unreleased resources
 - **Dangling pointers** – pointers that point to invalid memory



C Programming – Memory Management



Questions?



License

- This course (slides, examples, demos, videos, homework, etc.) is licensed under the "Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International" license



- Attribution: this work may contain portions from
 - "Programming Basics" course by Software University under CC-BY-SA license

Free Trainings @ Software University

- Software University Foundation – softuni.org
- Software University – High-Quality Education, Profession and Job for Software Developers
 - softuni.bg
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University @ YouTube
 - youtube.com/SoftwareUniversity
- Software University Forums – forum.softuni.bg

