UCSC Silicon Valley Extension

Advanced C Programming

Memory Management

Instructor: Radhika Grover

Overview

- Heap management functions: malloc, calloc, realloc, free
- How to use the functions to allocate and free memory
- Programming errors when using these functions and how to correct them

© SPEL Technologies, Inc. All rights reserved.

Heap management types

- Types:
  - void *: generic pointer to any type and can be converted to other pointer types
    - size_t : unsigned integer (can be a typedef of unsigned int, unsigned long, or unsigned long long on different systems)
- Heap management functions: malloc, calloc, realloc, free

© SPEL Technologies, Inc. All rights reserved.

Functions malloc and calloc

- void * malloc(size_t size)
  - Returns a pointer to the allocated memory of *size* bytes, OR
  - Returns NULL if memory could not be allocated.
  - The memory is not initialized.
  - Example, to allocate an array of *n* integers:

    int *ptr = (int *) malloc(sizeof(int) * n)

- void *calloc(size_t n, size_t size)
  - Allocates memory for an array of *n* elements of *size* bytes each
  - Returns a pointer to the allocated memory.
  - The memory is initialized to zero.
  - Example, to allocate an array of n integers and set each value to 0:

    int *ptr = (int *) calloc(n, sizeof(int))

© SPEL Technologies, Inc. All rights reserved.

Initializing memory

- calloc initializes memory to 0 but malloc leaves this task to programmer
- Can use memset in <string.h> to initialize memory:
  - void * memset(void *dest, int c, size_t count)

dest points to the block of memory to be written, *c* is the byte to write into this block, and *count* is the number of bytes to be set. Returns a pointer to *dest*.

© SPEL Technologies, Inc. All rights reserved.

Example: using malloc and memset

```c
// HeapMalloc/malloc_example.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(void)
{
	int newsize = 5;
	// malloc creates an array to store 5 ints
	int *arr = (int *)  malloc(sizeof(int) * newsize);

	if (arr == NULL) {
		// print memory could not be allocated
		printf("Error: could not allocate memory");
		return (-1);
	} else {
		// initialize all bytes in the array to 2
		memset(arr, 2, sizeof(int) * newsize);
		for (int i = 0; i < newsize; i++) {
			printf("%d\n", arr[i]); // print array
		}
		return 0;
	}
}
```


© SPEL Technologies, Inc. All rights reserved.

Function realloc

- void * realloc(void *ptr, size_t newsize)
  - Changes the size of the memory block to *newsize* bytes by allocating additional memory if needed.
  - Original contents of memory are not changed.
  - Newly allocated memory (if any) is uninitialized.
  - If ptr is not NULL, it must have been returned by an earlier call to malloc(), calloc(), or realloc() - why?

- Questions:
  - What happens if ptr is NULL?
  - What happens if *newsize* is 0?

© SPEL Technologies, Inc. All rights reserved.

Function realloc example

```
// HeapRealloc/realloc_example.c
int main(void)
{
	int *arr = NULL;
	int newsize = 5;
	int *ptr = realloc(arr, sizeof(int) * newsize); // realloc creates an array of size 5
	if (ptr == NULL) {
		printf("Error: could not allocate memory");
		return(-1);
	} else {
		arr = ptr; // important because arr points to NULL
		for (int i = 0; i < newsize; i++) {
			arr[i] = i; // put some values in the array
		}
		newsize = 10;
		ptr = realloc(arr, sizeof(int) * newsize); // realloc resizes the allocated array to 10
```


© SPEL Technologies, Inc. All rights reserved.

Function realloc example continued

```
    if (ptr == NULL) {
                printf("Error: could not allocate memory");
                return(-1);
} else {
                arr = ptr; // important
                for (int i = 0; i < newsize; i++) {
                    printf("%d\n", arr[i]); // print array
                }
                return 0;
            }
        }
}
```


© SPEL Technologies, Inc. All rights reserved.

Function free

- void free(void *ptr)
  - Frees memory space pointed to by ptr, which must have been returned by a previous call to malloc(), calloc(), or realloc(). (why?)
  - If free(ptr) has already been called before, undefined behavior occurs.
  - If ptr is NULL, no operation is performed.

© SPEL Technologies, Inc. All rights reserved.

Exercise: Find the errors

- Find the errors in the code segments:

1. // program creates a buffer to store 5 ints using malloc and store 10 at
   // each location
   int *arr = (int *) malloc(sizeof(int) * 5);
      arr[0] += arr[0] + 10;

2. // program resizes arr to store 5 ints using realloc
   int arr[1] = {1};
   int *ptr = realloc(arr, sizeof(int) * 5);
      strcpy(arr, 5);

3. // program reads the array size in argv[1] and creates an array of this size:
   char *arr = (char *) malloc(sizeof(char) * argv[1]);

© SPEL Technologies, Inc. All rights reserved.

Exercise: Solution

1.

2.

© SPEL Technologies, Inc. All rights reserved.

Exercise: Solution

```
3. // user gives the array size in argv[1]
   int *arr = (int *)  malloc(sizeof(int) * argv[1]);
      if (arr == NULL) {
            // print memory could not be allocated
      }
```

There are several problems here:

1. argv[1] must never be zero or negative, otherwise the behavior is undefined or may even lead to a buffer overflow

2. The product sizeof(int) * argv[1] must not create a value that is greater than the maximum value that is stored in a size_t (SIZE_MAX); Otherwise, the value will overflow resulting in a smaller number and the buffer that is created will have a smaller size than the size of data that will be stored in it. This can lead of a buffer overflow that may be exploited by a malicious user.
For example, if argv[1] is SIZE_MAX, the result sizeof(int) * SIZE_MAX will overflow a size_t.

© SPEL Technologies, Inc. All rights reserved.

Exercise: Solution

```
// HeapExercise/solution.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
      // Run the program with negative and large values of argv[1]; Add check for argc
    long long int len = atoll(argv[1]);
    if (len <= 0) {
        // print error that buffer size must be greater than 0
        printf("%lld Error: Buffer size must be greater than 0", len);
        return (-1);
    }
    else if (len > SIZE_MAX/sizeof(int)) {
        // print error message that the value of len may
        // lead to a buffer overflow
        printf("Error: %lld * sizeof(int) is greater than SIZE_MAX", len);
        return (-1);
    } else {
        // malloc creates an array to store len ints
        int *arr = (int *)  malloc(sizeof(int) * len);
```


© SPEL Technologies, Inc. All rights reserved.

Exercise: Find the errors

- This program reads a string in argv[1], stores it in a buffer created using malloc and displays it. Find all the errors in the program. (Hint: there are more than five errors.)

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

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

      size_t len = strlen(argv[1]);
      char *arr = (char *)  malloc(len + 1);
      strncpy(arr, argv[1], len);
      printf(arr);
      free(arr);
      return 0;
}
```


© SPEL Technologies, Inc. All rights reserved.

Exercise: Buffer overflow vulnerability solution

```
// HeapExercise2/solution.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
     // Add check for argc
    // calculate the length of the string
    size_t len = strlen(argv[1]);  // use strlen_s instead of strlen (why?)
        if (len > SIZE_MAX - 1 ) {
                // print error message that string size is greater than maximum allowed
                printf("Error: %zu + 1 is greater maximum allowed, buffer overflow may occur", len);
                return (-1);
        } else {
                // malloc creates an array to store a string of length len + 1 for null terminator
                char *arr = (char *)  malloc(len + 1);
                if (arr == NULL) {
                        // print memory could not be allocated
                        printf("Error: could not allocate memory");
                        return (-1);
                } else {
                        // use a safe function such as strncpy to copy string into arr and then free memory
                }
        }
}
```


© SPEL Technologies, Inc. All rights reserved.

Exercise: Freeing memory incorrectly

- What is the problem in the following program and how will you correct it?

```
// FreeMemory/problem.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
	char *arr = (char *)  malloc(10*sizeof(char));
	if (arr == NULL) {
		// print memory could not be allocated
		printf("Error: could not allocate memory");
		return (-1);
	} else {
		strcpy(arr, "Hello");
		free(arr);
		printf("%s", arr);
		return 0;
	}
}
```


© SPEL Technologies, Inc. All rights reserved.

Exercise: Freeing memory incorrectly solution

- Using strcpy is not a problem in the because the string is known (“Hello”) and fits in the buffer.

```
strcpy(arr, “Hello");
```

- The problem is that the contents of memory are being printed out after the free function are called.
- This can lead to a vulnerability where a malicious user can read data that should have been deleted.
- To prevent this, you should set the pointer to NULL after free is called.

```
free(arr);
arr = NULL;
```


© SPEL Technologies, Inc. All rights reserved.

Checking for memory leaks

- Use a tool to determine if heap memory (this is memory allocated with malloc, calloc, realloc) has been leaked .

- Valgrind is available for memory profiling and runs on several Linux systems. Visual Studio on Windows machines has a built in memory profiler.

- Download valgrind from http://valgrind.org and install it. You can run it on the command line for the executable Stack as:

- valgrind --tool=memcheck --leak-check=full --show-leak-kinds=all ./ Stack

© SPEL Technologies, Inc. All rights reserved.

Further reading

- Reference: https://en.cppreference.com/w/c/memory
- Robert Seacord. Secure Coding in C and C++.

© SPEL Technologies, Inc. All rights reserved.