Epitech Training

Session 7

C dynamic memory allocation

C dynamic memory allocation refers to performing manual memory management for dynamic memory allocation in the C programming language via a group of functions in the C standard library, namely malloc, realloc, calloc and free.

The C++ programming language includes these functions for compatibility with C; however, the operators new and delete provide similar functionality and are recommended by that language's authors.

Many different implementations of the actual memory allocation mechanism, used by malloc, are available. Their performance varies in both execution time and required memory.

C dynamic memory allocation

The C programming language manages memory statically, automatically, or dynamically. Static-duration variables are allocated in main memory, usually along with the executable code of the program, and persist for the lifetime of the program; automatic-duration variables are allocated on the stack and come and go as functions are called and return. For static-duration and automatic-duration variables, the size of the allocation must be compile-time constant (except for the case of variable-length automatic arrays). If the required size is not known until run-time (for example, if data of arbitrary size is being read from the user or from a disk file), then using fixed-size data objects is inadequate.

C dynamic memory allocation

The lifetime of allocated memory can also cause concern. Neither static- nor automatic-duration memory is adequate for all situations. Automatic-allocated data cannot persist across multiple function calls, while static data persists for the life of the program whether it is needed or not. In many situations the programmer requires greater flexibility in managing the lifetime of allocated memory.

These limitations are avoided by using dynamic memory allocation in which memory is more explicitly (but more flexibly) managed, typically, by allocating it from the free store (informally called the "heap"), an area of memory structured for this purpose. In C, the library function malloc is used to allocate a block of memory on the heap. The program accesses this block of memory via a pointer that malloc returns. When the memory is no longer needed, the pointer is passed to free which deallocates the memory so that it can be used for other purposes.

C dynamic memory allocation

The C dynamic memory allocation functions are defined in **stdlib.h**

- malloc
  - allocates the specified number of bytes
- realloc
  - increases or decreases the size of the specified block of memory. Reallocates it if needed
- calloc
  - allocates the specified number of bytes and initializes them to zero
- free
  - releases the specified block of memory back to the system

C dynamic memory allocation

Creating an array of ten integers with automatic scope is straightforward in C:

```
int array[10];
```

However, the size of the array is fixed at compile time. If one wishes to allocate a similar array dynamically, the following code can be used:

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

C dynamic memory allocation

Your computer always runs other processes at the same time. That’s why you should ALWAYS check that your computer still have available resources.

```c
int *array = malloc(10 * sizeof(int));
if (array == NULL) {
  fprintf(stderr, "malloc failed\n");
  return(-1);
}
```

When the program no longer needs the dynamic array, it should call free to return the memory it occupies to the free store:

```c
free(array);
```

C dynamic memory allocation

Common errors

The improper use of dynamic memory allocation can frequently be a source of bugs. These can include security bugs or program crashes, most often due to segmentation faults.

- Not checking for allocation failures
- Memory leaks
- Logical errors

C dynamic memory allocation

Not checking for allocation failures

Memory allocation is not guaranteed to succeed, and may instead return a null pointer. Using the returned value, without checking if the allocation is successful, invokes undefined behavior. This usually leads to crash (due to the resulting segmentation fault on the null pointer dereference), but there is no guarantee that a crash will happen so relying on that can also lead to problems.

C dynamic memory allocation

Memory leaks

Failure to deallocate memory using free leads to buildup of non-reusable memory, which is no longer used by the program. This wastes memory resources and can lead to allocation failures when these resources are exhausted.

C dynamic memory allocation

Logical errors

All allocations must follow the same pattern: allocation using malloc, usage to store data, deallocation using free. Failures to adhere to this pattern, such as memory usage after a call to free (dangling pointer) or before a call to malloc (wild pointer), calling free twice ("double free"), etc., usually causes a segmentation fault and results in a crash of the program. These errors can be transient and hard to debug – for example, freed memory is usually not immediately reclaimed by the OS, and thus dangling pointers may persist for a while and appear to work.

C - Buffer overflow

In computer security and programming, a buffer overflow, or buffer overrun, is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations.

Buffers are areas of memory set aside to hold data, often while moving it from one section of a program to another, or between programs. Buffer overflows can often be triggered by malformed inputs; if one assumes all inputs will be smaller than a certain size and the buffer is created to be that size, then an anomalous transaction that produces more data could cause it to write past the end of the buffer. If this overwrites adjacent data or executable code, this may result in erratic program behavior, including memory access errors, incorrect results, and crashes.

C - Buffer overflow

Exploiting the behavior of a buffer overflow is a well-known security exploit. On many systems, the memory layout of a program, or the system as a whole, is well defined. By sending in data designed to cause a buffer overflow, it is possible to write into areas known to hold executable code, and replace it with malicious code. Buffers are widespread in operating system (OS) code, so it is possible to make attacks that perform privilege escalation and gain unlimited access to the computer's resources. The famed Morris worm used this as one of its attack techniques.

Exercices

See session_7_ex.md on github.com for more details.