Seg-fault How-to

Nikolaus Correll

This is a mini tutorial on segmentation faults and memory leaks. C and C++ require you to do your own memory management. This usually works by allocating chunks of memory for your data and using pointers to accessing it. Likewise, you have to release memory after you are done using it.

Accessing data in a random memory location will result in a segmentation fault. Not releasing memory after you are using it will result in a memory leak.

A brief review on C syntax and memory allocation

The C compiler will reserve static memory for every variable that you declare. There are special types of variables, which are called pointers. A pointer is nothing more than a memory address (e.g. a 16 Bit value, depending on the memory model used). Thus, whether you declare variables such as integers, structures or pointers, memory will be allocated. Pointers, however, require you to allocate the memory where you actually store your data yourself.

Example 1: Pointer declarations

```
int i;
char* text=0;
char fixedstring[10];

struct
{
   int length;
   char* text;
}  string;

struct Matrix
{
   int rows, cols;
   int** data;
};

struct Matrix* rotation3D=0;
```

In this example, we are allocating memory for an integer “i”, a pointer to char “text”, a pointer to a char “fixedstring” with a chunk of memory holding 10 chars allocated to it at the same time, a structure “string” consisting of an integer and a pointer to char, and a pointer “rotation3D” to a previously defined structure “Matrix”. The ‘*’ after the variable type indicates that we are declaring a pointer. Similarly, two stars “**” declare a pointer to a pointer. The brackets ‘[]’ after the variable name indicate that we are declaring a pointer and statically allocate memory at the same time. This memory will be

automatically released after termination of the program and attempts to release it in your code will lead to a segmentation fault (similar to attempts to release memory for static variables such as “i").

Notice that the second structure is actually not declaring any memory but is defining a new data type (named “Matrix”). Only when declaring a variable using this data type, memory for two integers as well as for one pointer will be allocated. This pointer is a pointer (“*”) to a pointer to integer (“int*”). We will see why this makes any sense in the next example.

Example 2: Memory allocation

```
#include "example1.c"
#include <stdlib.h>

int main(){

   i = 10;
   text   = (char*) malloc(sizeof(char)*11);

   string.length = 12;
   string.text   = (char*) malloc(sizeof(char)*string.length);

   rotation3D = (struct Matrix*) malloc(sizeof(struct Matrix));
   rotation3D->rows=4;
   rotation3D->cols=4;
   rotation3D->data=(int**) malloc(sizeof(int*)*rotation3D->rows);

   for(i=0;i<rotation3D->rows;i++)
     rotation3D->data[i]=(int*) malloc(sizeof(int)*rotation3D->cols);

   free(text);
   free(string.text);

   for(i=0;i<rotation3D->rows;i++)
        free(rotation3D->data[i]);
   free(rotation3D->data);
   free(rotation3D);

   return 0;
 }
```

The function “malloc” (stdlib.h) allocates memory of a given size and returns a pointer to it and NULL otherwise (e.g. when you run out of memory). Here, the function “sizeof()” is helpful and works for standard data types as well those you have declared yourself (such as “struct Matrix”). Instead of calculating the size of the total array yourself, you can also use the function “calloc”.

Malloc will return a pointer to void (“void *”), which needs to be typecasted into the right pointer type. A typecast is achieved by preceding the variable to be type-casted by the desired datatype enclosed in brackets.

Remember that “rotation3D” is only a pointer to the Matrix structure defined in Example 1. We thus need to allocate memory for it during runtime. After memory has been allocated, we can access the structure’s fields using the “->” operator. For dynamically allocated structures, the “->” substitutes the “.” operator that is used for statically allocated structures (we are using this operator to access the “text” field in the “string” structure just a few lines above).

Now, it is getting tricky. The “data” field of our matrix is a pointer to a pointer. Our intention here is that we want to organize every row as an array with as many entries as there are columns. We thus allocate memory for as many pointers to integers as there are rows. We then loop through all these pointers and allocate for each an array with enough space for keeping an integer for every column.

Releasing memory is done using the function “free()”. We are first releasing the 11 bytes that have been allocated for “text”. We then release the 12 bytes that have been allocated for “string.text”. For the matrix, it will get messy again. If we would release the memory allocated to the “rotation3D” structure first, we would lose all references created for the “data” field – resulting in a memory leak. We thus need to release the memory in the opposite order as we have allocated it. First, we release the memory for storing the data of every row. We then release the array that stored the pointers to each row. Finally, we release memory for the “rotation3D” structure.

Exercise

Remove the line “rotation3D = (struct Matrix*) malloc…” and compile the program. Although the program compiles without errors, you will be getting a segmentation fault at run-time. Why?

Memory access is checked only at runtime. The compiler is not smart enough to detect that you are accessing unallocated memory when calling “rotation3D->rows=4” when you compile your code. Thus, the segmentation fault will only occur when you run the program.

Remove the for loop (two lines) that releases memory of the matrix data and compile the program. The program compiles without errors and you can even run it properly. Why bother then?

Again, memory access is only checked at runtime and the compiler won’t care that you do not properly release memory. Trouble starts then, when you do a lot of these operations and your program runs out of memory. For instance, when you don’t release images obtained from a video feed, or dynamically create data structures in a loop and systematically fail to release them.

Example 3: Advanced referencing and addressing

```
#include <stdio.h>

char string[12] = "Distributed";
char* index;
int i;
int* ip;

int main()
{
```

```
   printf("%s\n",string);
   for(i=0;string[i]!=0;i++) printf("%c",string[i]); printf("\n");

   for(index=string;(*index)!=0;index++) printf("%c",*index);
printf("\n");


i=10;
ip=&i;
*ip=11;

printf("%d\n",i);
return 0;
}
```

This example introduces advanced pointer manipulation and the “&” operator. The “&” operator yields the address of a variable, which can be used to create a pointer to this data.

We first allocate the string ‘Distributed’ statically and create a pointer on it. Notice that the word ‘Distributed’ has only 11 characters, the 12th is set to NULL (automatically), which allows the program to detect the end of a string (null-terminated). The program then outputs the string using three different methods of accessing the string array. First, we output the entire string using “printf”. Second, we output the string character by character using the “[]” operator. Third, we access the string using an extra pointer “index” that we move along the array. Notice that incrementing a pointer (index++) will automatically point it to the next data set of the array, independent of the size of its entries – the compiler will solve this for you.

The second part of this example introduces the “&” operator. We first obtain a pointer “ip” to the variable “i", by obtaining the address of “i" using “&”. We then change the data at the location of pointer “ip”. We then output the value of the variable. Indeed, the result is 11 and not 10.

Good practices

A segmentation fault usually occurs when you try to access data via pointers for which no memory has been allocated. It is thus good practice to initialize pointers with the value NULL, and set it back to NULL after the memory has been released. By checking whether the pointer is not NULL before accessing its data will avoid segmentation faults in most cases.

This is trickier when you are dealing with arrays, i.e. relative pointers. Then it is important that you check the array boundaries before each access to the array.

Things are getting very complicated when you are using 3rd party libraries and complicated data structures such as the Matrix structure from the examples. It is often unclear, where in the library memory is allocated and when it is released. In this case you can only track memory allocation with advanced tools such as valgrind.