







Course


-  ELEC-A7100

 Course materials

 Your points

 Microsoft Teams

 Code Vault
- 



This course has already ended.

« 2 Dynamic memory management

Course materials

4 Some memory management functions »

ELEC-A7100 / 5. Dynamic Memory / 3 Valgrind

Fifth round tasks as a [ZIP file](#).

Usually in the software projects there are situations where we are at the programming stage and cannot know how much memory the program needs in advance. In this case, we need a **dynamic memory** where size can be defined during the execution of the program. Dynamic memory is allocated by operating system upon request and generally it will always be free. This section gives introduction to computer memory organization, dynamic memory management and **valgrind** (tool is used to check memory leaks in the program).

Note: In this section, the tasks use valgrind tool to check memory leaks. If valgrind memory leaks or warnings are present in the code, only half of the total task point is given. If both valgrind and warnings exist, then you will get only quarter of the total task score. Valgrind information is provided later in this section.

Valgrind

Valgrind is a toolkit for analysing various aspects in a program, and commonly used for finding memory management errors. Valgrind will tell about memory leaks, i.e., memory blocks that have been allocated but not released, memory access errors (e.g., some cases of reading/writing invalid address), and so on.

Valgrind is used at program run time: first, a C program will be compiled and linked normally, to produce executable. Instead of normal execution, the executable can be run under Valgrind by typing valgrind name-of-executable on the command line. This will cause Valgrind to produce output about any memory errors it detects. If program is compiled with **-g** option, Valgrind will be able to point the line numbers related to suspected errors. More information and instructions about Valgrind memory checks can be found in their [web page](#). The 0th module also has some instructions on understanding the valgrind errors in the [valgrind instructions](#).

Valgrind is commonly available on Linux distributions, but unfortunately, its availability on Mac and Windows is limited, although some Mac versions support Valgrind.

We will now take a look for a couple of simple Valgrind error situations. Consider the following (badly implemented) function that will allocate some memory but the program will never release it:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  char *reverse_string(const char *str)
6  {
7      unsigned int len = strlen(str);
8      char *newstr = malloc(len + 1);
9      unsigned int i;
10     for (i = 0; i < len; i++) {
11         newstr[i] = str[len - i - 1];
12     }
13     newstr[i] = 0;
14     return newstr;
15 }
16
17 int main(void)
18 {
19     char *rev;
20     rev = reverse_string("Heippa vaan");
21     printf("%s\n", rev);
22
23     rev = reverse_string("Moikka vaan");
24     printf("%s\n", rev);
25
26     return 0;
27 }
```

When you compile and run this program, it will work produce expected output. However, let's see, what happens when this program is run using valgrind.

```
==9== Memcheck, a memory error detector
==9== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==9== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==9== Command: ./revstring.exe
==9==
naav appieH
naav akkiom
==9==
==9== HEAP SUMMARY:
==9==      in use at exit: 24 bytes in 2 blocks
==9==    total heap usage: 3 allocs, 1 frees, 4,120 bytes allocated
==9==
==9== 12 bytes in 1 blocks are definitely lost in loss record 1 of 2
==9==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==9==    by 0x108704: reverse_string (revstring.c:8)
==9==    by 0x108765: main (revstring.c:20)
==9==
==9== 12 bytes in 1 blocks are definitely lost in loss record 2 of 2
==9==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==9==    by 0x108704: reverse_string (revstring.c:8)
==9==    by 0x108781: main (revstring.c:23)
==9==
==9== LEAK SUMMARY:
==9==    definitely lost: 24 bytes in 2 blocks
==9==    indirectly lost: 0 bytes in 0 blocks
==9==    possibly lost: 0 bytes in 0 blocks
==9==    still reachable: 0 bytes in 0 blocks
==9==    suppressed: 0 bytes in 0 blocks
==9==
==9== For counts of detected and suppressed errors, rerun with: -v
==9== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Valgrind prints errors.

The above **valgrind** results show that there is a memory leak in the program. **12 bytes in 1 blocks are definitely lost** means that the program has dynamically allocated 12 bytes of memory, which has not been released, and thus there is a memory leak. Valgrind memory leak results shows the funtion names, line number from which the function is called and lines with malloc function. You should keep track of function names and line numbers, which are printed in this context. In this program, Memory leak has occurred twice, because the main function called the **reverse_string** twice.


Task 5.1: Memory leak

Correct the above program so that there are no memory leaks in the program, (i.e), release memory allocated from a suitable point in the program when it is no longer needed. The program will continue to print the same output of two reversed strings.

Points **10 / 10** My submissions **2**

Deadline Friday, 9 July 2021, 19:59

To be submitted alone

 This course has been archived (Saturday, 31 December 2022, 20:00).

Reverse String

Select your files for grading

 **main.c**

No file chosen

Valgrind categorizes memory leak errors in different categories on the basis of whether the reserved memory areas addresses yet safe. Sometimes freeing memory is not possible even when the code for releasing memory is properly implemented, for example if the pointer to the earlier allocated memory is lost.

Next is an another example where we have own strcat implementation - a function that adds one string after another. In contrast to the original version, memory for destination string is dynamically allocated in this implementation. Try to run the program on your own computer.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  char *mystrcat(char *dest, const char *src)
6  {
7      char *origdest = dest;
8      while(*dest) {
9          dest++;
10     }
11
12     while (*src) {
13         *dest++ = *src++; // Copies character and increases/moves pointer
14     }
15     *dest = 0;
16
17     return origdest;
18 }
19
20 int main(void)
21 {
22     char *str = malloc(7);
23     strcpy(str, "Aatami");
24
25     str = mystrcat(str, "Beetami");
26     printf("%s\n", str);
27     free(str);
28
29     return 0;
30 }
```

The program may once again seem do the right thing (or it might crash too), but if you run the program using Valgrind, you will notice that there are memory leaks in the program.

Valgrind captures different kind of memory violations. Above is one example:

Invalid write of size 1, error is followed by line number and exact error message "Adress xxx is N bytes after a block of size Y alloc'd". This means that the program is written over the memory area, which was not allocated for or by the program. This error is repeated several times as the program is doing invalid write many times in the program.

Task 5.2: My Strcat

Correct the above **mystrcat** function so that there must not be any valgrind errors present in the program. The problem with the original implementation is that the memory allocated for the destination string is not big enough to store result of the concatenation of two strings.


It is fairly common that Valgrind shows repeated errors that refer to same lines. Try to concentrate on the first error of the repetitive errors. Once you find the reason for the error, many of the other error lines should go away.

More complete summary of different valgrind error messages can be found in the 0th module

Points **10 / 10** My submissions **6**

Deadline Friday, 9 July 2021, 19:59

To be submitted alone

 This course has been archived (Saturday, 31 December 2022, 20:00).

My Strcat

Select your files for grading

 **source.c**

No file chosen

Task 5.3: Dynamic Array

In this task, we practice dynamic memory allocation and extending arrays

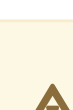
Two functions need to be implemented in this task. To collect points, some kind of implementation must exist for both functions. Remember, valgrind errors must not be present to get full points.

- Implement function **int *create_dyn_array(unsigned int n)** that allocates an int array for n integers. n is given as argument to the function when it is called. After allocating the array, the function should read the given number of integers to the array from user, using the **scanf** function. After the right amount of integers have been read, the function returns pointer to the dynamically allocated array.
- Implement function **int *add_dyn_array(int *arr, unsigned int num, int newval)** that adds a single integer to the existing dynamically allocated array of integers (arr). The length of the existing array is num, and the new integer to be added is given in parameter newval. You'll need to ensure that array is extended to hold the additional integer. Check that the function works correctly when called several consecutive times.

Points **30 / 30** My submissions **1**

Deadline Friday, 9 July 2021, 19:59

To be submitted alone

 This course has been archived (Saturday, 31 December 2022, 20:00).

Dynamic Array

Select your files for grading

 **source.c**

No file chosen

« 2 Dynamic memory management

Course materials

4 Some memory management functions »