Valgrind

Introduction

All programs use memory resources during execution. Some resources are statically defined (such as int, double and float variables in functions), these memory resources are statically allocated to the stack and will be freed when exiting a function (except for global and static variables). Thus dynamic allocation of a memory resource is needed where such resources will reside in a place called the head heap (each program has its own stack and heap, both are given by the OS when the program is loaded).

You have seen this in Java before, when you create a **new** instance of a class, you can use this instance anywhere in your code as long as you have a reference to it. In C you will use a function called **malloc** to allocate memory resources on the heap.

What's the big fuss is all about?

Every program needs memory resources in order to work properly and obtain maximal functionality. So why do we make a big fuss about it?
The answer to this is simple; **MEMORY RESOURCES ARE LIMITED**.

Every machine has limited memory resources, and when running many programs this resource will drain out. For this reason, the operating system will usually limit the amount of memory each program can use. Hence we need to make sure that resources that are no longer needed must be freed.

Why did you hear about this just until now?
Because you have developed java programs. In java you have a special tool called garbage collector which does the job for you. In C however, when you no longer need a resource you must free it using **free()** function.

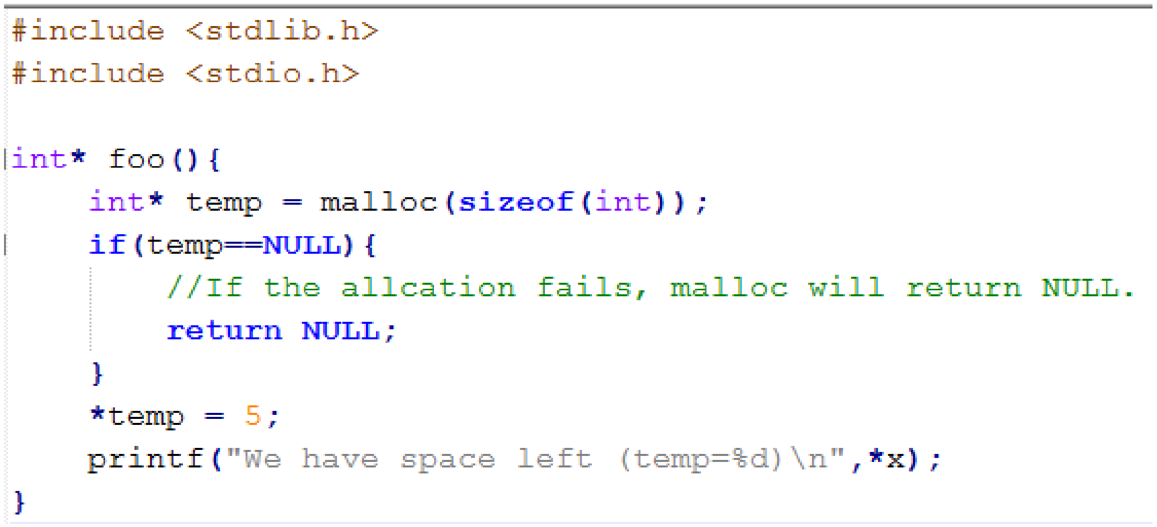
What is Valgrind

Valgrind is a special tool that has many functionalities. The most important one is that it can track all memory resources used by your program. We will use it to track if any resource was used but not freed. This is called memory leak!

Example

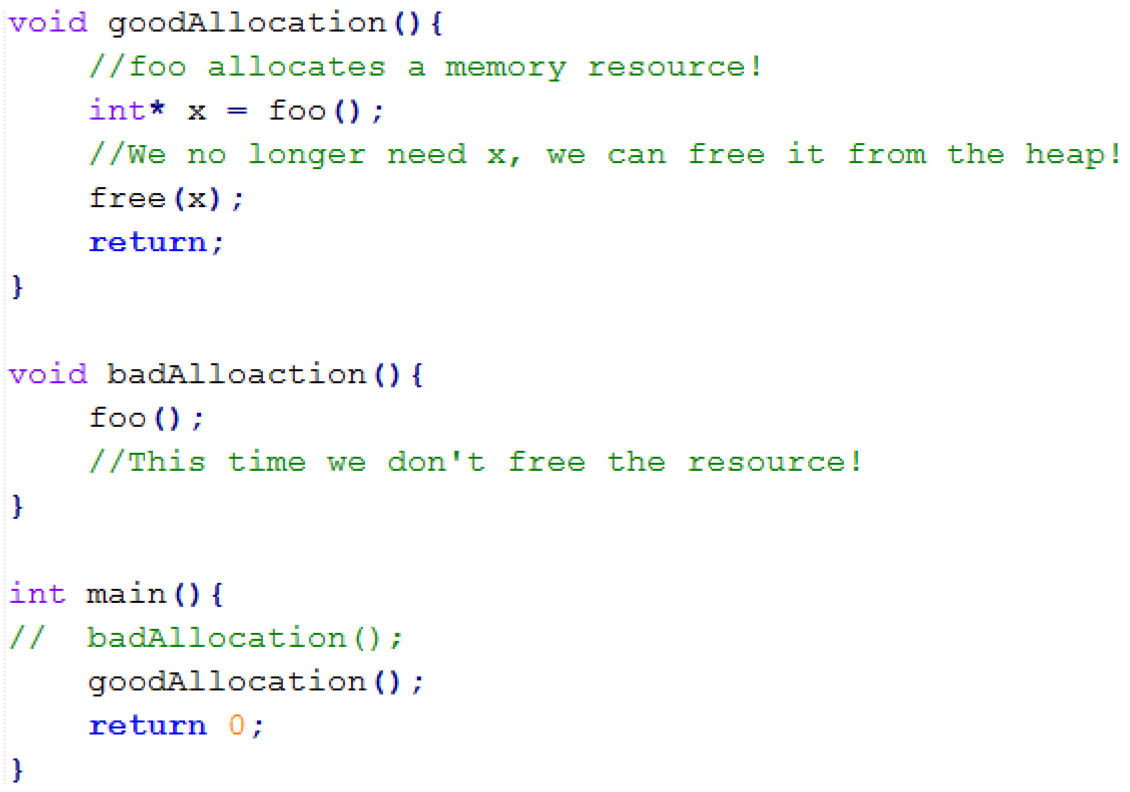
Let us look at the following program (main.c):

We see that foo() allocates an int variable, but doesn't free it. So whoever calls foo() must make sure to free the memory resource pointed by temp.



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

int* foo(){
    int* temp = malloc(sizeof(int));
    if(temp==NULL){
        //If the allcation fails, malloc will return NULL.
        return NULL;
    }
    *temp = 5;
    printf("We have space left (temp=%d)\n",*x);
}
```

The rest of the code is shown below (main.c):



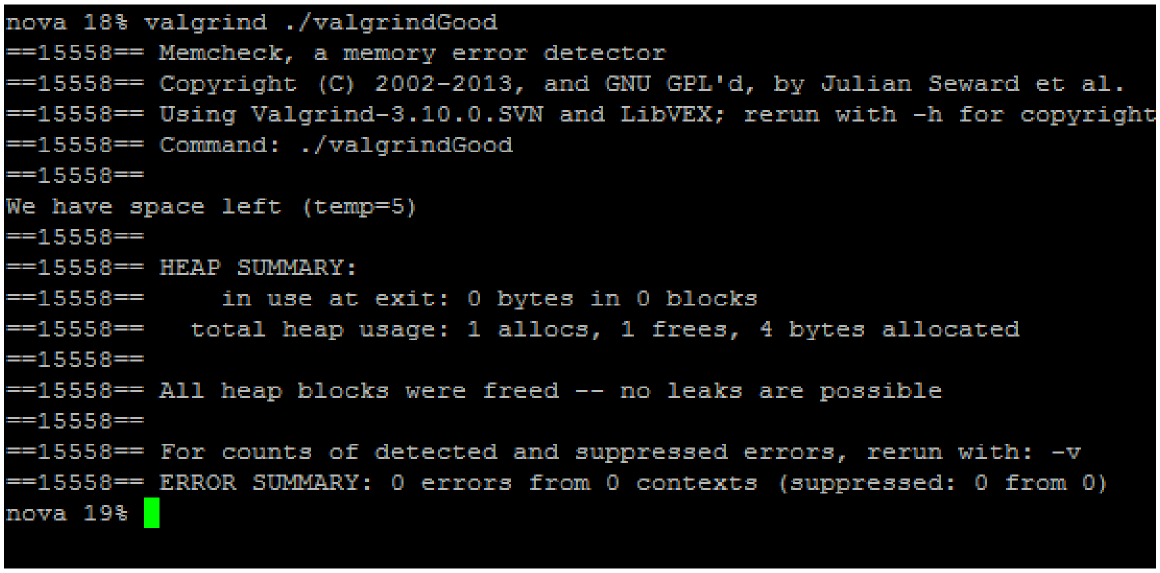
```c
void goodAllocation(){
    //foo allocates a memory resource!
    int* x = foo();
    //We no longer need x, we can free it from the heap!
    free(x);
    return;
}

void badAlloaction(){
    foo();
    //This time we don't free the resource!
}

int main(){
//  badAllocation();
    goodAllocation();
    return 0;
}
```

In the code above we see two functions: goodAllocation() and badAllocation(). Both functions uses foo(), but goodAllocation() uses free to clean the memory used and badAllocation() "forgets" to do so.

To use valgrind and detect this problem, we first need to compile our code into an executable program. After compilation (in our example we called our program **valgrindGood**) we write the following command:

```
>> valgrind ./valgrindGood
```

After compiling the above code and running it with valgrind (note that badAllocation() is commented out – it is not called) we get the following message:



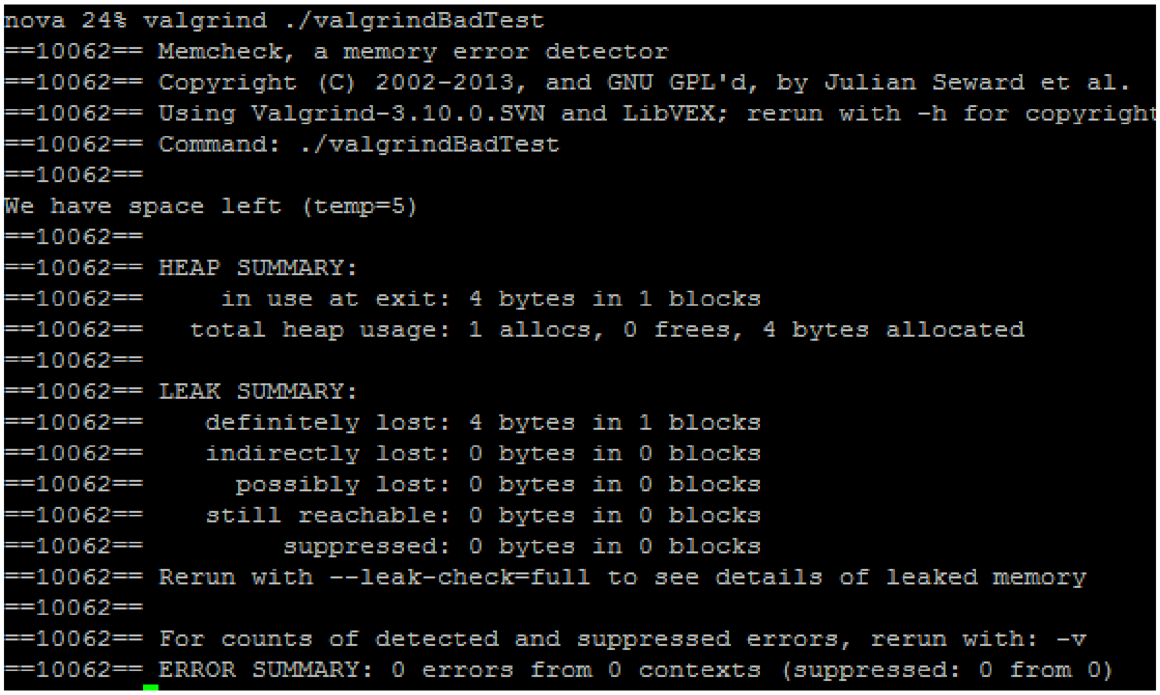
```
nova 18% valgrind ./valgrindGood
==15558== Memcheck, a memory error detector
==15558== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==15558== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright
==15558== Command: ./valgrindGood
==15558==
We have space left (temp=5)
==15558==
==15558== HEAP SUMMARY:
==15558==     in use at exit: 0 bytes in 0 blocks
==15558==   total heap usage: 1 allocs, 1 frees, 4 bytes allocated
==15558==
==15558== All heap blocks were freed -- no leaks are possible
==15558==
==15558== For counts of detected and suppressed errors, rerun with: -v
==15558== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
nova 19%
```

This indicates that we have used **4 bytes (an integer variable uses 4 bytes in the memory)** and all the memory used was freed at exit. **No problem!**

Now we comment out the call for function **goodAllocation()** and call **badAllocation()** instead. Our new main function will look like this:

```
int main(){
     badAllocation();
//   goodAllocation();
     return 0;
}
```

Now we compile the code again and make a new executable **valgrindBadTest.** Let us look at the message printed out by valgrind:

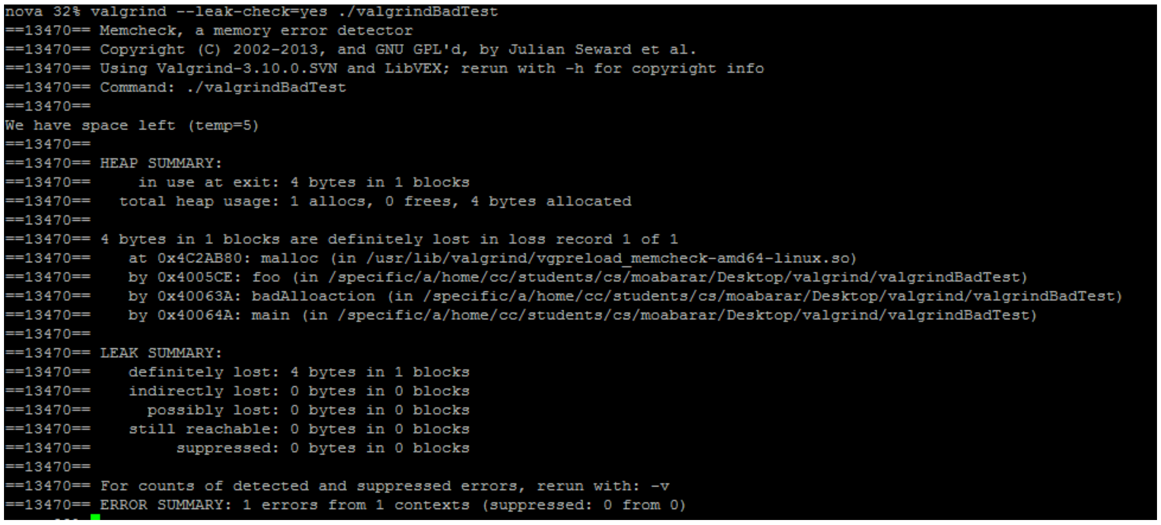


```
nova 24% valgrind ./valgrindBadTest
==10062== Memcheck, a memory error detector
==10062== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==10062== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright
==10062== Command: ./valgrindBadTest
==10062==
We have space left (temp=5)
==10062==
==10062== HEAP SUMMARY:
==10062==     in use at exit: 4 bytes in 1 blocks
==10062==   total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==10062==
==10062== LEAK SUMMARY:
==10062==    definitely lost: 4 bytes in 1 blocks
==10062==    indirectly lost: 0 bytes in 0 blocks
==10062==      possibly lost: 0 bytes in 0 blocks
==10062==    still reachable: 0 bytes in 0 blocks
==10062==         suppressed: 0 bytes in 0 blocks
==10062== Rerun with --leak-check=full to see details of leaked memory
==10062==
==10062== For counts of detected and suppressed errors, rerun with: -v
==10062== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

We can see that the message got bigger and we now have a new message LEAK SUMMARY. This means **we have memory leak**.

To get extra information you can use --leak-check=yes flag that show you extra information to help you trace your problem. Let us run the following command:

>> valgrind --leak-check=yes ./valgrindBadTest



```
nova 32% valgrind --leak-check=yes ./valgrindBadTest
==13470== Memcheck, a memory error detector
==13470== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==13470== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==13470== Command: ./valgrindBadTest
==13470==
We have space left (temp=5)
==13470==
==13470== HEAP SUMMARY:
==13470==     in use at exit: 4 bytes in 1 blocks
==13470==   total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==13470==
==13470== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==13470==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==13470==    by 0x4005CE: foo (in /specific/a/home/cc/students/cs/moabarar/Desktop/valgrind/valgrindBadTest)
==13470==    by 0x40063A: badAlloaction (in /specific/a/home/cc/students/cs/moabarar/Desktop/valgrind/valgrindBadTest)
==13470==    by 0x40064A: main (in /specific/a/home/cc/students/cs/moabarar/Desktop/valgrind/valgrindBadTest)
==13470==
==13470== LEAK SUMMARY:
==13470==    definitely lost: 4 bytes in 1 blocks
==13470==    indirectly lost: 0 bytes in 0 blocks
==13470==      possibly lost: 0 bytes in 0 blocks
==13470==    still reachable: 0 bytes in 0 blocks
==13470==         suppressed: 0 bytes in 0 blocks
==13470==
==13470== For counts of detected and suppressed errors, rerun with: -v
==13470== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

We see the trace where the lost block was defined. [You can see from the picture above that it was called in **main->badAllocation->foo->malloc**].