lab_memory

Malevolent Memories

Due: Feb 03, 23:59 PM

Doxygen | Lab handout | Handout solution | Lab slides

Assignment Description

In this lab, you will learn about the memory checking utility `valgrind`.

The first utility you will learn about is Valgrind. Valgrind will help you detect memory errors and practice implementing the big three. Valgrind is an extremely useful tool for debugging memory problems and for fixing segfaults or other crashes. This lab is also particularly important because **we will be checking for memory errors and leaks on your assignments**. You will lose points for memory leaks and/or memory errors (we will also teach you the difference between a memory leak and a memory error). You should check your code with Valgrind before handing it in. You should also be aware that Valgrind will only detect a leak if your program allocates memory and then fails to deallocate it. It cannot find a leak unless the code containing the leak is executed when the program runs. Thus, you should be sure to test your code thoroughly and check these tests with Valgrind.

☢ Valgrind does not work well or at all on later versions of macOS! In particular, we have noticed Valgrind always reporting memory leaks.

We strongly recommend working on this assignment on an EWS system!

Lab Insight

As you can tell from the description, this lab will be heavily focused on memory management and good practices when dealing with memory management issues such as memory leaks. It is even possible to build lightweight tools similar to Valgrind, where these tools can trace the memory used by the program and report information related to memory management issues such as memory leaks. If this lab is interesting for you, CS 241 is a course that covers both how to write memory allocation models as well as how to develop tools similar to Valgrind in functionality.

Code Description

For this lab, you will be fixing bugs in course staff's Student-To-Room allocation program. Since many CS classes are very large (CS 225 has nearly 800 students!), exams are usually spread across several rooms. Before each exam, course staff have to allocate different students to different rooms, so that everyone can take the test with enough space.

For example, if there were only two classrooms of equal size, students in the first half of the alphabet (last name letters A - N) might go to Siebel 1404, while students in the second half of the alphabet (letters M - Z) might go to DCL 1320.

However, with more rooms, this problem becomes more difficult. In the sample situation provided, there are 9 classrooms for the exam, varying in seating capacity from 43 to 70 seats (i.e. 21 to 35 students seated every-other-desk). Although we'll have to break up the alphabet more, we'd still like to assign students with the same first letter of their last name to the same room, as this makes going to the right room easier.

We've provided you the code to solve this problem, however, it has several memory bugs in it. You'll have to use Valgrind, as well as some debugging skills from lab_debug, to find the bugs and fix them. Note, **there are no bugs in the `fileio` namespace**.

A reference for the lab is provided for you in Doxygen form.

Checking Out the Code

After reading this lab specification, the first task is to check out the provided code from the class repository.

To check out your files for the third lab, run the following command in your `cs225git` directory:

```
$
```

```
git fetch release
git merge release/lab_memory -m "Merging initial lab_memory files"
```

If `git` happens to complain about unrelated histories, use this command:

```
$
```

```
git fetch release
git merge release/lab_memory --allow-unrelated-histories -m "Merging initial lab_memory
files"
```

This should update your directory to contain a new directory called `lab_memory`.

Background on Valgrind

Valgrind is a useful tool to detect memory errors and memory leaks.

Valgrind is a free utility for memory debugging, memory leak detection, and profiling. It runs only on Linux systems. To prepare your project to be examined by Valgrind you need to compile and to link it with the debug options `-g` and `-O0`. Make sure your `Makefile` is using these options when compiling. In order to test your program with Valgrind you should use the following command:

```
$
```

```
valgrind ./yourprogram
```

To instruct valgrind to also check for memory leaks, run:

```
$
```

```
valgrind --leak-check=full ./yourprogram
```

You will see a report about all found mistakes or inconsistencies. Each row of the report starts with the process ID (the process ID is a number assigned by the operating system to a running program). Each error has a description, a stack trace (showing where the error occurred), and other data about the error. It is important to eliminate errors in the order that they occur during execution, since a single error early could cause others later on.

Here is a list of some of the errors that Valgrind can detect and report. (Note that not all of these errors are present in the exercise code.)

- **Invalid read/write errors.** This error will happen when your code reads or writes to a memory address which you did not allocate. Sometimes this error occurs when an array is indexed beyond its boundary, which is referred to as an "overrun" error. Unfortunately, Valgrind is unable to check for locally-allocated arrays (i.e., those that are on the stack.) Overrun checking is only performed for dynamic memory.

Example

```
int * arr = new int[6];
arr[10] = -5;
```

- **Use of an uninitialized value.** This type of error will occur when your code uses a declared variable before any kind of explicit assignment is made to the variable.

Example

```
int x;
cout << x << endl;
```

- **Invalid free error.** This occurs when your code attempts to delete allocated memory twice, or delete memory that was not allocated with `new`.

Example

```
int * x = new int;
delete x;
delete x;
```

- **Mismatched `free() / delete / delete []`.** Valgrind keeps track of the method your code uses when allocating memory. If it is deallocated with different method, you will be notified about the error.

Example

```
int * x = new int[6];
delete x;
```

- **Memory leak detection.** Valgrind can detect three sources of memory leakage.
  - A **still reachable block** happens when you forget to delete an object, the pointer to the object still exists, and the memory for object is still allocated.
  - A **lost block** is a little tricky. A pointer to some part of the block of memory still exists, but it is not clear whether it is pointing to the block or is independent of it.

- A **definitely lost block** occurs when the block is not deleted but no pointer to it is found.

Example

```
int * x = new int[6]; // no corresponding delete x
```

More information about the Valgrind utility can be found at the following links:

- http://www.valgrind.org/docs/manual/quick-start.html
- http://www.valgrind.org/docs/manual/faq.html#faq.reports
- http://www.valgrind.org/docs/manual/manual.html

Fixing Memory Bugs (using Valgrind)

Before fixing the bugs, you'll need to compile the code:

```
$
```

```
make
```

This will create an executable file called allocate, which you can run with:

```
$
```

```
./allocate
```

You can then run Valgrind on allocate:

```
$
```

```
valgrind ./allocate
```

This works fine for fixing the memory errors, however, to fix the memory leaks, you'll need to add `--leak-check=full` before `./allocate`:

```
$
```

```
valgrind --leak-check=full ./allocate
```

Once you have fixed all the Valgrind errors, you can test your program output using:

```
$
```

```
./allocate > output.txt
diff output.txt soln_output.txt
```

Note that most of the work in this lab consists of fixing Valgrind's errors and memory leaks, rather than the program's output, which should be correct once the memory errors are fixed.

Below is sample code and the corresponding valgrind output to give an idea of the output that vlagrind will give with certain errors.

Example Code used:

```
12  int main()
13  {
14     int * arr = new int[10];
15         int * x = new int;
16         int * y;
17         arr[0] = *y; // y has not be initialized
18         delete arr; // Wrong delete, should be delete[]
19         delete x;
20         delete y; // Should not delete, not on heap
21     return 0;
22  }
```

Valgrind output:

```
==22153== Use of uninitialised value of size 8
==22153==    at 0x4007B9: main (main.cpp:17)
==22153==
==22153== Mismatched free() / delete / delete []
==22153==    at 0x4C2B1CD: operator delete(void*) (vg_replace_malloc.c:576)
==22153==    by 0x4007E0: main (main.cpp:18)
==22153==  Address 0x5a22040 is 0 bytes inside a block of size 40 alloc'd
==22153==    at 0x4C2A8E8: operator new[](unsigned long) (vg_replace_malloc.c:423)
==22153==    by 0x40079D: main (main.cpp:14)
==22153==
==22153== Conditional jump or move depends on uninitialised value(s)
==22153==    at 0x40080F: main (main.cpp:20)
==22153==
==22153== Conditional jump or move depends on uninitialised value(s)
==22153==    at 0x4C2B180: operator delete(void*) (vg_replace_malloc.c:576)
==22153==    by 0x400820: main (main.cpp:20)
==22153==
==22153== Invalid free() / delete / delete[] / realloc()
==22153==    at 0x4C2B1CD: operator delete(void*) (vg_replace_malloc.c:576)
==22153==    by 0x400820: main (main.cpp:20)
==22153==  Address 0x400830 is in the Text segment of
/home/pacole2/TA/examples/prog
==22153==    at 0x400830: __libc_csu_init (in /home/pacole2/TA/examples/prog)
==22153==
==22153==
==22153== HEAP SUMMARY:
==22153==     in use at exit: 0 bytes in 0 blocks
==22153==   total heap usage: 2 allocs, 3 frees, 44 bytes allocated
==22153==
==22153== All heap blocks were freed -- no leaks are possible
==22153==
==22153== For counts of detected and suppressed errors, rerun with: -v
==22153== Use --track-origins=yes to see where uninitialised values come from
==22153== ERROR SUMMARY: 5 errors from 5 contexts (suppressed: 0 from 0)
```

The number, 22153, represents the process id (PID) given to the program by the operating system. Below is a break down of each error.

- The first error, originating from line 17, is due to the fact that y was never initialized
- The second error comes from the fact that the wrong delete is being used in line 18. From looking at the declaration of `arr` we can see that `delete[]` should've been used instead.
- The last error occurs on line 20 when trying to delete something that wasn't initialized or pointing to the heap.

- **Stack buffer overflow.** This error occurs when you go out of bounds within an array created on the stack.

Example

```
int arr[100];
arr[1] = 0;
arr[5 + 100]; //NOPE
```

- **Use after return.** This error occurs when you return a stack variable at the end of a function and try to use it after it's out of scope.

Example

```
int * arr1;
void func() {
   int arr2[100];
   arr1 = arr2;
}
int main() {
   func();
   return arr1[10]; //NOPE
}
```

- **Double free or Invalid free.** Double free occus when you free an heap object twice. Invalid free's occur when you free a non-heap object.

Example

```
int * arr = new int[100];
delete [] arr;
delete [] arr; //NOPE
```

```
int arr[100];
delete [] arr; //NOPE
```

Further Testing

To test the code further, you may use the provided test program to see whether your program runs fine without any memory leaks. We expect that your code produces no memory leaks for both the allocate program and the test program. To compile the test program, please run the following:

```
$
```

```
make test
```

You can test whether the test program successful works without any memory issues by running

```
$
```

```
valgrind ./test
```

If you notice no errors or memory leak errors outputted from Valgrind, your test program has successfully run as well. If not, it's time to Valgrind your way through the code once more :)

To get more specific issues about memory leak releated issues with Valgrind, you may get more information by using:

```
$
```

```
valgrind --leak-check=full ./test
```

GDB: A Debugger

While Valgrind tells you what went wrong in your program after it has been executed, GDB is a debugging tool that allows you to see what is going on 'inside' your program WHILE it executes! It can also be used after your program has crashed for debugging purposes as well. In particular, Valgrind works well for memory related errors, but GDB can handle crashes from those as well as other classes of bugs, such as logic errors. To learn how to use GDB for your lab and mps, visit this page:

Guide: Learn GDB

Grading Information

The following files are used to grade this assignment:

- `allocator.cpp`
- `allocator.h`
- `letter.cpp`
- `letter.h`
- `room.cpp`
- `room.h`

All other files, including `main.cpp` and any testing files you have added **will not** be used for grading.

Guide: How to submit CS 225 work using git