lab_memory Malevolent Memories

Due: Sunday, September 13 at 11:59 PM

Doxygen for lab_memory

Assignment Description

In this lab, you will learn about two memory checking utilities: **AddressSanitizer** (aka ASAN) and **Valgrind**.

As mentioned above, the first utility you will learn about is AddressSanitizer. ASAN will help you detect memory errors and practice implementing the big three. ASAN is an extremely useful tool for debugging memory problems, and can be faster than gdb (GNU debugger) for fixing segfaults or other crashes. This lab is also particularly important because **ASAN will be used to grade your MPs**. 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 ASAN before handing it in. You should also be aware that ASAN 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 ASAN.

Background on ASAN

ASAN checks for a wide variety of things. Here are some things that may pop up when debugging:

- **Out-of-bounds access to heap, stack, and globals.** This error occurs when you allocate some memory and then try to access a region outside your allocated space.

  Example

  ```C++
  int * arr = new int[100];
  return arr[100];
  ```

- **Use-after-free (dangling pointer reference).** This error occurs when you try to use memory you have already freed. It is especially helpful when you have several pointers referring to the same memory.

  Example

```cpp
int * arr = new int[100];
delete [] arr;
return arr[5]; //NOPE
```

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

Example

```cpp
int * arr = new int[100];
arr[0] = 0;
int ret = arr[5 + 100]; //NOPE
delete [] arr;
return ret;
```

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

Example

```cpp
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

```cpp
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

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

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

- **Memory leak detection.** ASAN 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.

Fixing Memory Bugs (using ASAN)

Before fixing the bugs, you'll need to compile the code. ASAN terminates the program upon the first invalid memory access. So, if you have an invalid memory access, you'll have to fix it before moving on to other errors. This incremental procedure should help you step through memory bugs one at a time.

To compile the code for the lab, run

```TERMINAL
make
```

(In this and future assignments, `make` will produce 2 versions of each executable: the "normal" version and a version that has ASAN enabled.)

This will create two executable files named `allocate` and `allocate-asan`. Run the ASAN version with

```TERMINAL
./allocate-asan
```

which will check for both memory errors and leaks.

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

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

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

Using Valgrind (optional)

In the past, Valgrind has been a useful tool to detect memory errors and memory leaks. Valgrind does what ASAN does but because it doesn't make the memory checking code into the binary executable, it has to interpret the code during runtime. Therefore, Valgrind can be slower than ASAN. Nonetheless, if you are interested in how it works, you can read on about Valgrind. NOTE: You cannot used Valgrind and ASAN at the same time. Make sure to run it only on the non-ASAN executable!

Background on Valgrind

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
```
TERMINAL

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

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

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;
  ```
  C++

- **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

  ```C++
  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

  ```C++
  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

  ```C++
  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.

    Example

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

  - 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.

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
```
TERMINAL

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

```
./allocate
```
TERMINAL

You can then run Valgrind on allocate:

```
valgrind ./allocate
```
TERMINAL

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
```
TERMINAL

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

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

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.

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 `cs225` directory:

```
svn update
```
TERMINAL

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

Code Description

For this lab, you will be fixing bugs in course staff's Student-To-Room allocation program. Since CS 225 is a large class, exams are usually spread across several rooms. Before the exam, we 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 ASAN, 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.

Committing the Code

To commit your code, first create and add a `partners.txt` file:

```
gedit partners.txt
svn add partners.txt
```
TERMINAL

Then commit:

```
svn ci -m "lab_memory submission"
```
TERMINAL

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.