ALGORITHMS LABORATORY: AVOIDING PROGRAMMING ERRORS

Overview

We'll be covering some common programming mistakes that can take up a lot of the allotted time - which can be put to better use for thinking on the algorithmic solution to the problem at hand, specially for your lab test.

One of the most dreaded errors is the segmentation fault, often leaving no clues about what went wrong. We cover a small part with walkthrough example of the gdb debugger to handle segmentation faults quickly, as well as general info about some common instances that may cause such faults.

Core Dumped? - What went wrong and where? - Using the GDB Debugger

We'll do a walkthrough considering the tree.cc file. It contains a simple modified BST implementation with only insert functionality. The insert function is modified as:

i) If key to be inserted is not already in tree, insert it as in regular BST
ii) If a node with that key exists, then compare the abs difference with node's left and right child values - and proceed with insertion to the subtree rooted at the child which had lower difference (in case of equal diff., choose any).

If you go on to compile and run this program, well - you get segmentation fault.

- Print statements to the rescue?

Many of us resort to print statements to see where it went wrong. If you open and see the tree.cc file, the first statement prints "Starting", but it is not printed to the output!

This is a problem with stdout stream buffering, and the short solution is to use \n with your print string, but to get an idea of what's actually happening, you can refer to this stackoverflow post.

The point is, printing is not the fastest or the optimal solution along with sometimes being unreliable as well. We now turn to see if the debugger can provide better answers.

- Compilation with debugging symbols

The gdb debugger works with the debugging symbols linked into a program, which with g++ and gcc can be achieved using the -g flag.
Compilation with debugging symbols:

```
$ g++ -g tree.cc                                          (gcc for c file)
```

This creates the executable file a.out for tree.cc with debugging symbols.

- Running the debugger

To run the debugger with your executable loaded,
```
$ gdb a.out
```

This opens up the debugger, printing some information,
```
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
....
....
Reading symbols from a.out...done.
(gdb)
```

To run the program with gdb,
$ run

We get the segfault again, but this time with more information:

```
.. in TreeNode<int>::getKey (this=0x0) at tree.cc:19
19      return key;
```

So, we now know the segfault occurs at the 19th line where we return the key for a node. Cool! But, this doesn't help that much - we want to at what step the error occurred. We will now print the stack of calls just before the segfault using "bt" (backtrace) command in the debugger:

```
(gdb) bt
```
which gives:

```
#0  .. in TreeNode<int>::getKey (this=0x0) at tree.cc:19
#1  .. in Tree<int>::insert_ (this=0x615030, n=0x6150f0,
    key=10) at tree.cc:48
#2  .. in Tree<int>::insert_ (this=0x615030, n=0x615050,
    key=10) at tree.cc:45
#3  .. in Tree<int>::insert_key (this=0x615030, key=10)
    at tree.cc:69
#4  .. in main () at tree.cc:95
```

Well, this is a tell-tale. #3 tells us that insert_key with key=10 is called, which calls #2 insert_ with key=10 and #1 at line number 48, which is executed only in case of existing key in tree, there is segfault when getting keys for the children.

The error obviously is that a node may not have both/any children, and applying getKey on a null node ptr will get you a seg fault. Cool!

Sometimes, the variable values are not so easy to get when loops are involved. For this, we use breakpoints, where the program breaks and you can see each statement it executes step by step. Just to see how it works, we will set breakpoint at line 68 which we know from the bt calls the insert_ that finally calls a faulty getKey().

```
(gdb) break 68
```

It would be inconvenient to break at every call. We know that problem is when the key argument is 10, so we add the condition:

```
(gdb) condition 1 key==10
```

So the breaking happens only when insert_key is called with 10. This is super-useful as it'll help you see what the differences are in insertion of 10 the first and the second time - allowing you to arrive at the error more swiftly.

```
(gdb) run
```

You'll now see:

```
Breakpoint 1, Tree<int>::insert_key (this=0x615030, key=10) at
tree.cc:69
69      root = insert_(root, key);
```

Now, you need to use the cmd "step", and then keep pressing enter (which basically repeats the last gdb cmd i.e. step):

```
(gdb) step
Tree<int>::insert_ (this=0x615030, n=0x615050, key=10) at tree.cc:41
41      if(n == NULL)
(gdb)
44      if(key > n -> getKey())
(gdb)
TreeNode<int>::getKey (this=0x615050) at tree.cc:19
19      return key;
(gdb)  ...

and so on.
```

To quit gdb, just enter 'q', followed by 'y' if prompted.
While this much familiarity should be good enough to deal with problems in your lab test and any upcoming assignments, there is more that can be found with gdb man page ($ man gdb).

Other General Errors:

User gives input n, and an array of that size has to be made

This is a common scenario across many lab problems. One very common mistake that has been found is:

```
int n;
scanf("%d", &n);
int arr[n];
```

The reason this is wrong is that static arrays need to be allocated when you compile the program (gcc/g++) and before runtime (./a.out). Since at that point value of n is not know, the array boundary is ambiguous.
While this may fortuitously work (which is not good for you, since you'll be unaware of any errors, while running program at a different time can lead to segfault/stack smashing), this is the classic example of undefined behavior in c/c++ (refer this). The reason this may work is same as the reason the following can work some times without error:

```
int a[20];
a[1000] = 1;
printf("%d", a[1000]);          (prints 1 in some cases, segfault/stack smashing in others)
```

To use arrays whose size is not known at compile time, you need to use dynamic allocation. Static allocation should be used only when you know the size of the array before you start compiling your program!

***** stack smashing detected *****

This occurs when there is an illegal memory accesses to stack-allocated data/ unallocated area. For example, consider the snippets:

```
Snippet 1                    Snippet 2

char c[5];                   int j[3];
scanf("%s", c);              j[0] = 1;
// user enters-> abcdef      j[3] = 4;
```

For 1, the user entered string tries to write beyond the stack allocated space to c, and in 2, the program tries to do the same by trying to write to j[3], both of which will give you the stack smashing errors.

Allocating massive static arrays:
Tempted to use static arrays instead of dynamically allocated ones, some may define arrays of max size like 10^8 or 10^9 in the main function (on stack). While it is difficult to say the exact limit, but assigning a static array or size near these is a sure shot seg fault. In these cases, while you can allocate it out of main to use heap memory - it is discouraged since using globals is not a good practice in general and can cost you some marks. Use dynamic allocations on user given input size for such arrays.

Uninitialised dynamic arrays:

```
char* c;
....
c[2] = 'a';                  (-> seg fault)
```

scanf: Missing & with scanf or having it when scanning string

C++ Strings Initialization

```
string q;
q[0] = 'a'; q[1] = 'b';
cout << q;                   (-> empty output, q is initialized as 0 length string)
```

Correct way is:

```
q += 'a'; q += 'b'
```

or if known beforehand, simply:

```
string q = "ab";
```

C++ Strings Null Character Termination
Changing behavior in different versions, but almost never reliable.

```
string q = "hello";
q[2] = '\0'
cout << q.length();          (-> output is 5)
cout << q;                   (-> output is helo)
```