Testing

Golden rules of debugging

- Understand the requirement
- Make it fail (writing test cases for different scenario)
- Simplify the test case
- Read the right error message
- Check the environment
- Divide and conquer

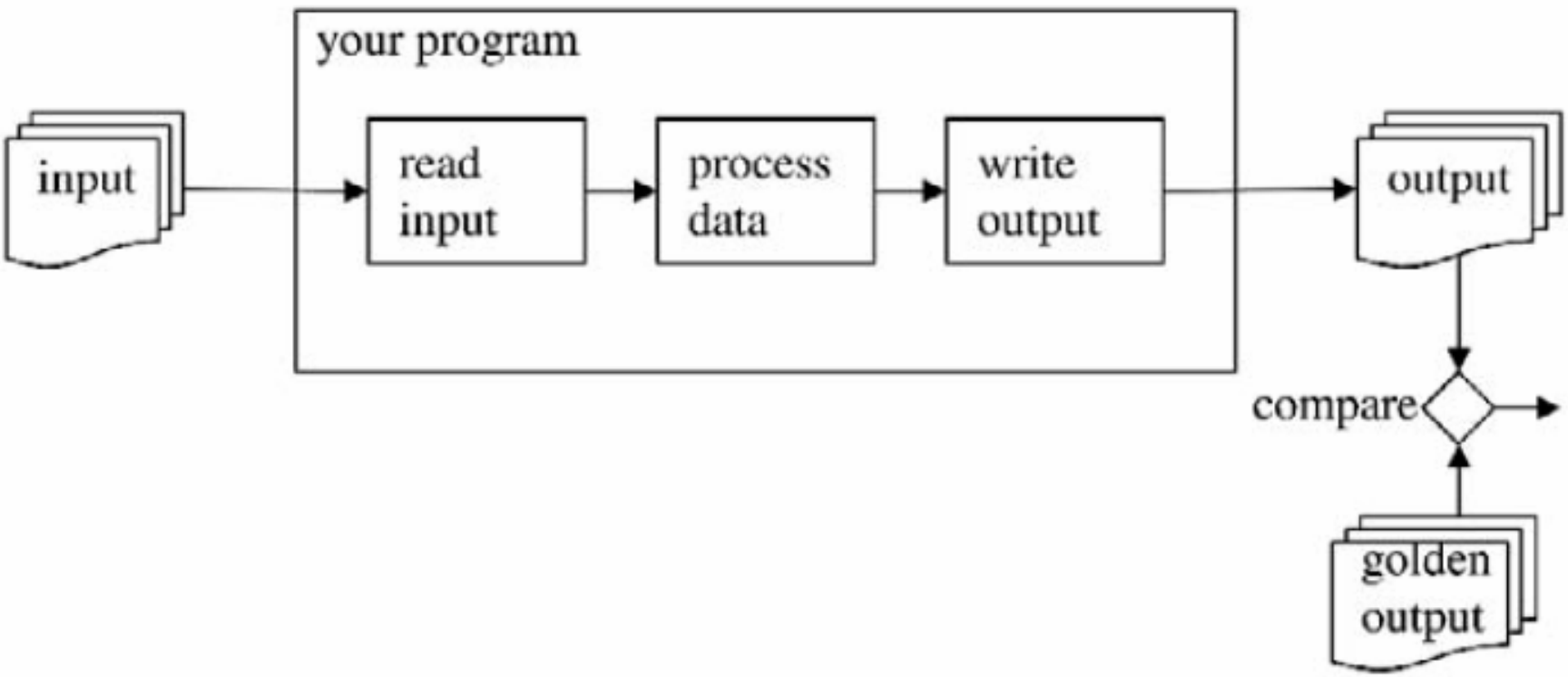
Golden rules of debugging

- Match the tool to the bug
- One change at a time
- Keep the audit trail
- Get a fresh view (second opinion)
- If you didn’t fix it, it ain’t fixed
- Cover your bug fix with a regression test

Regression test

- Regression test means “check whether yesterday’s functionality is still working well today”
- Put known test case together
- The test should be automated and allow frequent and efficient execution
- The test should be self-checking
- It can also be memory test or performance regression test

System test


your program
input
read input
process data
write output
output
compare
golden output

Your workshop tools

- Functional issues : A source code debugger
- Memory issues: A memory debugger
- Performance issues: A profiler

Meet the bug family

- Common bug
- Sporadic bug (not easy to repeat)
- Heisengbugs comes from Heisenberg's uncertainty principle
  - The bug was gone when turned debugging info
  - Added printf and it work, removed then fail

Usual cause of Heisenbugs?

- Race conditions
- Memory outlaw such as access violations, reading uninitialized variables, dangling pointers, array bound violations

  dangling pointers are pointers that do not point to valid object
- Compiler optimization

Meet the bug family

- Bugs hiding behind bugs
- Secret bugs
  - It happens with your customer, but due to confidentiality, you cannot have the their input files/ running environments

Memory issues

- Memory leaks (allocate but do not deallocate the memory)
- Incorrect use of memory management
  - using malloc to allocate but delete to deallocate
- Buffer overruns
  - Memory outside the allocated boundaries is over written
- Uninitialized memory bugs
  - Using uninitialized memory

Debugger

- A.k.a source code debugger or symbolic debugger

- Need to compile program with debug information (symbolic information)

Debug example

```
1   /* factorial.c */
2   #include <stdio.h>
3   #include <stdlib.h>
4
5   Int factorial(int n) {
6   int result = 1;
7   if(n == 0)
8   return result;
9   result = factorial(n-1) * n;
10 return result;
11 }
12
13 int main(int argc, char **argv) {
14 int n, result;
15 if(argc != 2) {
16 fprintf(stderr, "usage: factorial n, n >=0\n");
17 return 1;
18 }
19 n = atoi(argv[1]);
20 result = factorial(n);
21 printf("factorial %d = %d\n", n, result);
22 return 0;
23 }
```

GCC compilation

- To compile gcc with debug information use: -g

> gcc -g -o factorial factorial.c

Running program with gdb


```
> gdb factorial
<lots of copyright stuff..>
(gdb) run 1
<messages about Loaded symbols..>
factorial 1 = 1
Program exited normally
```

Running program with gdb

(gdb) run -1

Program received signal SIGSEGV, Segmentation fault.

factorial (n=-103583) at factorial.c:6

6 int result = 1;

Stack tracing on a program crash

(gdb) backtrace

#0 <..> in factorial (n=-105582) at factorial.c:9

<lots of frames..>

#103581 <..> in factorial (n=-2) at factorial.c:9

#103582 <..> in factorial (n=-1) at factorial.c:9

#103583 <..> in main (argc=2, argv=0x761ce8) at factorial.c:20

You can navigate up and down stack using up and down command

Breakpoints

- Line breakpoint
  pause at specific line
- Function breakpoint
  pause at the first line of the function
- Conditional breakpoint
  pause once a certain condition holds
- Event breakpoint
  pause if certain events occur

Navigate through program

Basic debugger command:

- run: will start the program
- start: run program until the first line of main()
- pause: interrupt running program
- continue: resume execution

Navigate through program

- step-into (step): go to the next executable line of code. In case of function call, it will go to the first line of the function call
- step-over (next): go to the next executable line of code in the same call stack level
- step-out (finish): go down one level in the stack

Inspect data

- print: print the current value of a variable

- display: the value gets updated whenever the program is paused

Debug session on factorial 13

13! = 6227020800 where we get 1932053504

```
> gdb factorial
...
(gdb) start 13
(gdb next
...
(gdb) next
20 result = factorial(n);
(gdb) print n
$1 = 13
(gdb) step
factorial (n=13) at factorial.c:6
6 int result = 1;
```

Debug session

(gdb break 8
Breakpoint 2 at <..>: file factorial.c, line 8
(gdb) break 10
Breakpoint 3 at <..>: file factorial.c, line 10
(gdb) continue
Continuing.
Breakpoint 2, factorial (n=0) at factorial.c:8
(gdb) print n
$1 = 0
(gdb) print result
$2 = 1
(gdb) display result
1: result = 1

Debug session

```
(gdb) continue
Breakpoint 3, factorial (n=1) at factorial.c:10
1: result = 1
(gdb) continue
...
Breakpoint 3, factorial (n=12) at factorial.c:10
1: result = 479001600
(gdb) continue
Breakpoint 3, factorial (n=13) at factorial.c:10
1: result = 193053504
(gdb) print 13 * 479001600
$5 = 193053504
```

GDB running process

```
#include <stdio.h>
int a,b=0;
int main(void){
  printf(“a: “);
  if(scanf(“%d”,&a)<1) return 0;
  printf(“a = %d \n”,a);
  printf(“b: “);
  if(scanf(“%d”,&b)<1) return 0;
  printf(“a=%d, b=%d\n”,a,b);
  return 0;
}
```

GDB running process

gcc –o ab ab.c
$./ab
a: 123
a=123
b: _
$ps aux | grep ab
roland 21601 0.0 0.0 3648 456 pts/11 ./ab
roland 21665 0.0 0.0 5132 672 pts/12 grep ab
$gdb
(gdb) attach 2601

GDB running process

(gdb) where

#0 0x8000dd2970 read () from /lib/libc.so.6

#1 0x8000d80b40 IO_file_underflow() from /lib/libc.so.6

#2 0x8000d66903 IO_vfscanf() from /lib/libc.so.6

#3 0x8000d7245c scanf() from /lib/libcs.so.6

#4 0x8000d2f1a0 main() from /lib/libcs.so.6

GDB running process

(gdb) frame 5 change stack frame to frame 5

(gdb) disassemble $pc

0x80000d2f1a0 : test %eax, %eax
0x80000d2f1a2 : jle 0x40058c <scanf>
0x80000d2f1a4 : mov 0x2003fe(%rip),%edx
0x80000d2f1aa : mov 0x2003fc(%rip),%esi
0x80000d2f1b0 : mov 0x40068f,%edi
0x80000d2f1b5 : xor %eax, %eax
0x80000d2f1b7 : call 0x400df8 <printf>

GDB running process

(gdb) x/w 0x60097c

(print word data from this memory location)

0x60097c : 7b

(gdb) x/w 0x600978

0x60097c : 00

(gdb) set *(int *) 0x60097c = 1234567

(gdb) continue

Advance GDB

insert_sort 12 5 19 22 6 1

result:

1

5

6

12

19

22

Pseudo code

```
Main()
set y array to empty   (output array)
call get args (get num_inputs  input x[i])
call process data
for i = 1 to num_inputs
call insert(x[i])
new_y = x[i]
find first y[i] for which new_y < y[i]
call scoot_over(j)
      shift y[j], y[j+1], … to the right to make room for new y
set y[i] = new_y
```

C Code

```
1  #include <stdio.h>
2  int x[10],y[10],num_inputs,num_y=0;
3
4  void get_args(int ac,char **av){
5          int i;
6          num_inputs = ac-1;
7          for(i=0;i<num_inputs;i++)
8                  x[i] = atoi(av[i+1]);
9  }
10
11 void scoot_over(int jj){
12         int k;
13         for(k=num_y-1;k>jj;k++)
14                 y[k] = y[k-1];
15 }
16 void insert(int new_y){
17         int j;
18         if(num_y=0){
19                 y[0] = new_y;
20                 return;
21         }
22         for(j=0;j<num_y;j++){
23                 if(new_y<y[j]){
24                         scoot_over(j);
25                         y[j] = new_y;
26                         return;
27                 }
28         }
29 }
```

C Code

```
30 void process_data(){
31
    for(num_y=0;num_y<num_inputs;num_y++)
32              insert(x[num_y]);
33 }

34 void print_results(){
35      int i;
36      for(i=0;i<num_inputs;i++)
37              printf("%d\n",y[i]);
38 }

39 int main(int argc,char **argv){
40      get_args(argc,argv);
41      process_data();
42      print_results();
43 }
```

Compile and run

gcc -g -Wall -o insert_sort ins.c

insert_sort 12 5

<loop forever>

Gdb

gdb -tui insert_sort

- break 18

check whether it keeps calling here or not

Fix to == sign

Bug #2

insert_sort 12 5

result:

5

0

Gdb

- break 22
- print y

First iteration

- print y

Second iteration

GDB

Something is wrong with function scoot_over

- break 13

GDB

- Is k start at the right position?

k = num_y

Bug #3

insert_sort 12 5

Segmentation fault

GDB

At line 14

- print k

(show large number, shouldn't it stop)

k--

Bug #4

insert_sort 12 5 19 22 6 1

result:

1
5
6
12
0
0

Homework

Memory debugger can use for:

- Memory leaks
- Accessing memory that was already freed
- Freeing memory that was never allocated
- Mixing C malloc/free with new/delete
- Using delete instead of delete []
- Array out-of-bound errors
- Accessing memory that was never allocated
- Uninitialized memory read
- Null pointer read or write

Example 1: Memory leaks

```
#include <stdio.h>

void test(){
  char * string;
  string = malloc(10);
}
void test2(){
  char * string;
  string = malloc(10);
  free(string);
}

int main(){
  int i,j;
  for(j=0;j<10000;j++){
    for(i=0;i<100000;i++){
     test();
     test2();
    }
  }
  printf("Finish \n");
  return 1;
}
```

What's wrong with this code?

Using top command

```
top - 16:40:08 up 1 day,  5:36,  7 users,  load average: 0.08, 0.02, 0.01
Tasks: 132 total,   2 running, 129 sleeping,   1 stopped,   0 zombie
Cpu(s): 45.0%us,  5.0%sy,  0.0%ni, 50.0%id,  0.0%wa,  0.0%hi, 0.0%si,  0.0%st
Mem:   2035512k total,  1751640k used,   283872k free,    59256k buffers
Swap:  2104472k total,   164948k used,  1939524k free,   498068k cached

PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+ COMMAND
30605 pop      20   0 1106m 1.1g  264 R   99 55.6   0:06.52 test
30606 pop      20   0  2356 1008  760 R    0  0.0   0:00.01 top
  1      root      20   0  1940  280  252 S    0  0.0   0:00.70 init
```

Example 2: Detecting Memory Access Errors

```
1   /*   main.c  */
2    #include <stdio.h>
3    Int main(int argc, char *argv[]){
4    const in size=100;
5    Int n,sum=0;
6    Int *A= (int *) malloc (sizeof(int)*size);
7
8    for(n=size;n>0;n--)
9     A[n] = n;
10  for(n=0;n<size;n++)
11    sum += A[n];
12  printf("sum = %d \n",sum);
13  return 0;
14  }
```

Running under Valgrind

- gcc –g main.c
- Valgrind –tool=memcheck –leak-check=yes ./main

Detecting Uninitialized Memory Reads

```
==11323== Use of uninitialised value of size 4
==11323==    at 0x1BA429B7: (within /lib/tls/libc.so.6)
==11323==    by 0x1BA46A35: _IO_vfprintf (in .../libc.so.6)
==11323==    by 0x1BA4BDAF: _IO_printf (in .../libc.so.6)
==11323==    by 0x804855C: main (main1.c:12)
==11323==
==11323== Conditional jump or move depends on
==11323== uninitialised value(s)
==11323==    at 0x1BA429BF: (within .../libc.so.6)
==11323==    by 0x1BA46A35: _IO_vfprintf (in .../libc.so.6)
==11323==    by 0x1BA4BDAF: _IO_printf (in .../libc.so.6)
==11323==    by 0x804855C: main (main1.c:12)
```

sum += A[n]; // error for element 0

Detecting memory leaks

```
==11323== 400 bytes in 1 blocks are definitely lost
==11323==    in loss record 1 of 1
==11323==    at 0x1B903F40: malloc
==11323==    (in /usr/lib/valgrind/vgpreload_memcheck.so)
==11323==    by 0x80484F2: main (main1.c:6)
```

Report at the end of the program

Example 3: memory allocation

```
1   /* main2.c */
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include <string.h>
5   int main(int argc, char* argv[]) {
6   char* mystr1=strdup("test");
7   char* mystr2=strdup("TEST");
8   mystr1=mystr2;
9
10 printf ("mystr1=%s\n", mystr1);
11 free(mystr1);
12
13 printf ("mystr2=%s\n", mystr2);
14 free(mystr2);
15 return 0;
16 }
```

Compile and run the program

```
> gcc -g main2.c
> valgrind --tool=memcheck --leak-check=yes ./a.out
```

Invalid read address

```
==11787== Invalid read of size 4
==11787==    at 0x1BA71903: strlen (in .../libc.so.6)
==11787==    by 0x1BA4BDAF: _IO_printf (in .../libc.so.6)
==11787==    by 0x8048554: main (main2.c:13)
==11787==  Address 0x1BB26060 is 0 bytes inside a block
==11787==  of size 5 free'd
==11787==    at 0x1B9040B1: free (in ...memcheck.so)
==11787==    by 0x8048541: main (main2.c:11)
```

Mystr1 was freed

Deallocate

```
==11787== Invalid free() / delete / delete[]
==11787==    at 0x1B9040B1: free (in ...memcheck.so)
==11787==    by 0x8048562: main (main2.c:14)
==11787==  Address 0x1BB26060 is 0 bytes inside a block
==11787==  of size 5 free'd
==11787==    at 0x1B9040B1: free (in ...memcheck.so)
==11787==    by 0x8048541: main (main2.c:11)
```

Mystr2 was already deallocated

Memory leak

```
==11787== 5 bytes in 1 blocks are definitely lost in
==11787== loss record 1 of 1
==11787==    at 0x1B903B7C: malloc (in ...memcheck.so)
==11787==    by 0x1BA7163F: strdup (in .../libc.so.6)
==11787==    by 0x8048504: main (main2.c:6)
```

It has never been deallocated

Combining memory and source debugger

```
> valgrind --tool=memcheck --leak-check=yes \
  --db-attach=yes ./a.out
```

When to use memory debugger

- When porting to new OS
- When your program crashes
- When beginning to debug a strange bug
- As an integral part of your regression tests
- When program is too slow than it should

Memory profiling

- Check that there are no memory leaks
- Estimate the expected memory use
- Measure memory consumption over time with multiple inputs
- Find the data structures that consume memory

Example: allocating array

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define blocksize 1024

void wait_for_input(const char *prefix, int is_interactive) {
    char c;
    if(is_interactive) {
        printf("%s hit return to continue\n", prefix); fflush(stdout);
        c = getchar();
    }
    else
    {   sleep(1); }
}

int main(int argc, char **argv) {
    const char *usage = "usage: testmalloc i[interactive]|n n iterations\n";
    int n, i, j, iterations, is_interactive = 0;
    int **myarray;
    if(argc != 4) {
        fprintf(stderr, usage);
        return 1;
    }
```

Example: allocating array

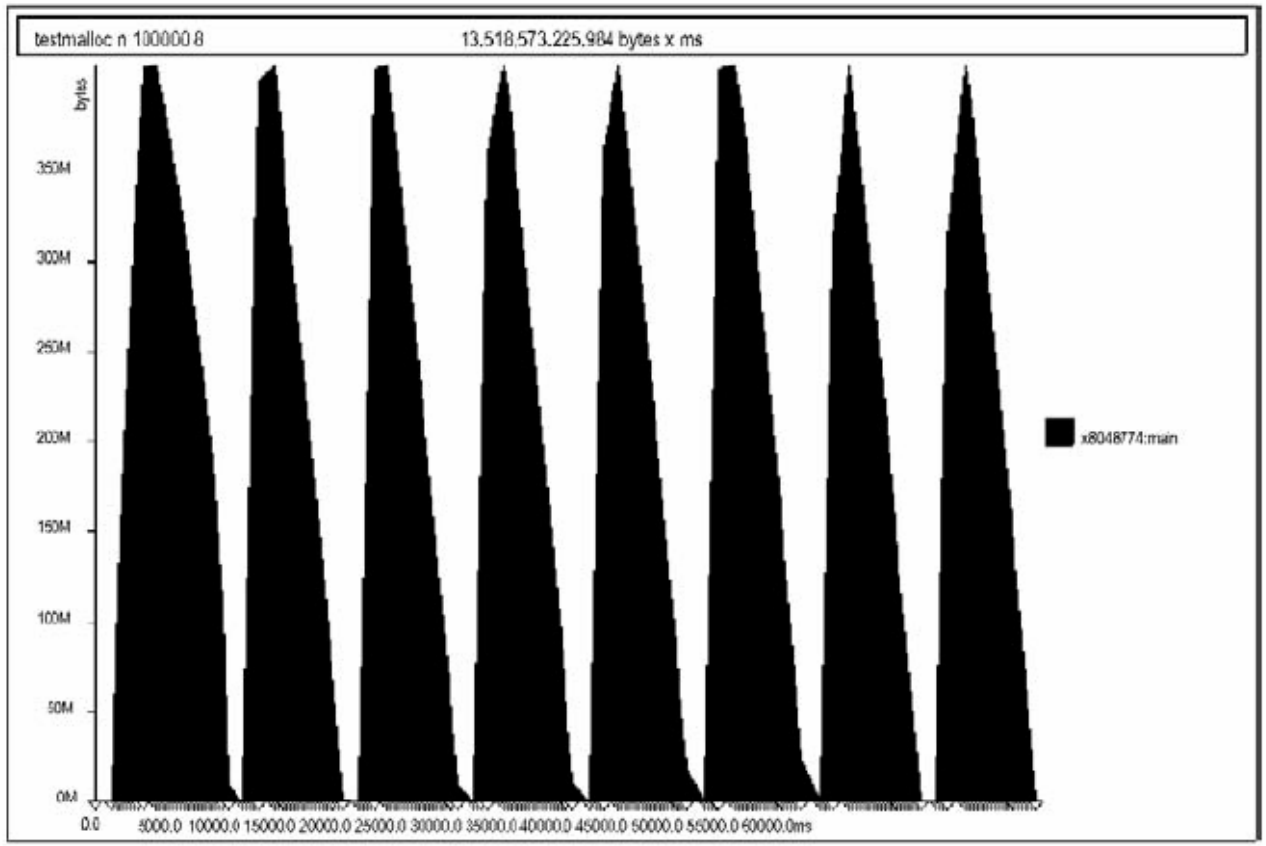

```
    if(argv[1][0] == 'i')
        is_interactive = 1;

    n = atoi(argv[2]);
    iterations = atoi(argv[3]);
    if(n <= 0 || iterations < 0) {
        fprintf(stderr, usage);
        return 2;
    }

    for(i=0; i<iterations; i++) {
      wait_for_input("before malloc: ", is_interactive);
        myarray = (int **) malloc(n * (sizeof(int *)));
      for(j=0; j<n; j++) {
            myarray[j] = (int *) malloc(blocksize * sizeof(int));
        }
        wait_for_input("after malloc:  ", is_interactive);
        for(j=0; j<n; j++) {
            free(myarray[j]);
        }
        free(myarray);
    }
    return 0;
}
```

Memory profiler

> valgrind --tool=massif ./testmalloc n 100000 8

ms_print massif.out.##### > output

Massif output using ms_print



Called from:

84.9% : 0x8048773: main (testmalloc.c:53)

3.9% : 0x8048746: main (testmalloc.c:47)

Example : genindex

genindex: download from esl.ait.ac.th with input1.txt

genindex is a program to read text files and printout an index of word in the text file. The index is a sort listed of unique word.

Memory profiling

- Check that there are no memory leaks
- Estimate the expected memory use
- Measure memory consumption over time with multiple inputs
- Find the data structures that consume memory

Estimate the expected memory use

Data structure use:

typedef map<string,list<int>,less<string> > WordIndexType // assume double link list pointers + 1 integer (12 bytes)

WordIndexType wordindex;

vector<string> lines;

Assume average string size is 4, then there will be n/4 strings.

Total memory size = n+ 12*n/4 = 4*n

Memory profiling

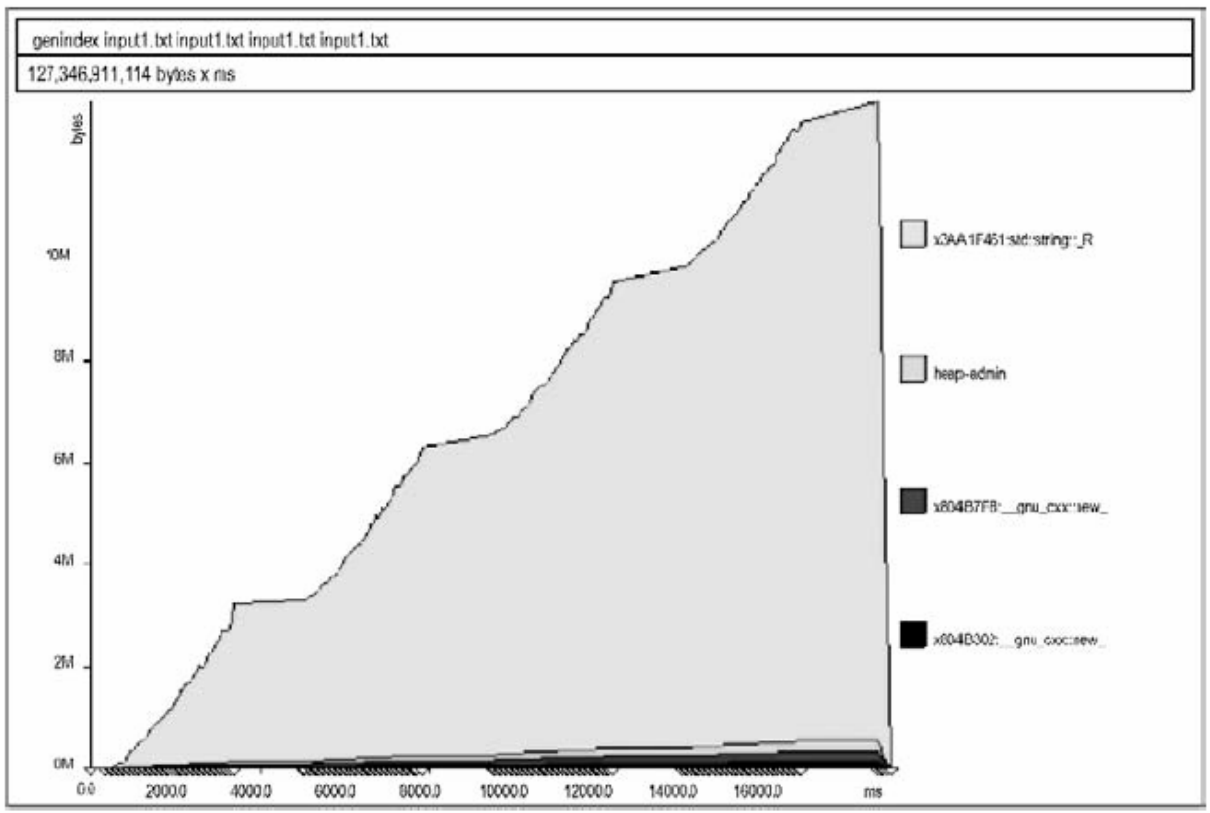
```
> g++ -g -o genindex genindex.cc
> valgrind --tool=massif \
  ./genindex input1.txt input1.txt input1.txt input1.txt >log
```

result:

```
Called from:
  93.0% : 0x3AA1F460: std::string::_Rep::_S_create(unsigned,
          unsigned, std::allocator<char> const&)
          (in /usr/lib/libstdc++.so.6.0.6)
```

Massif output

Memory use increases to 2.3 M bytes per 20K byte input file



Total memory size

```
-- memory size for index of 'input1.txt' file size=20016:
--     filename=14 wordindex=47211 lines=2340979 total=2388204
...
-- memory size for all data structures: 9552816 bytes
```

It requires total of almost 10 Mb instead of 100 K

Source code

```
105 int FileIndexType::scan_file(char *fname) {
...
112 string buffer;
...
119 while(1) {
120 c = getc(fp);
121
122 if(c == EOF || c == '\n') {
123 add_to_index(newword, current_line);
124 newword = "";
125 current_line++;
126 lines.push_back(buffer);
...
133 else if(c == ' ' || c == '\t' || c == '\r') {
...
137 buffer = buffer + (char) c;
138 }
139 else {
...
141 buffer = buffer + (char) c;
142 }
143 filesize++;
144 }
```

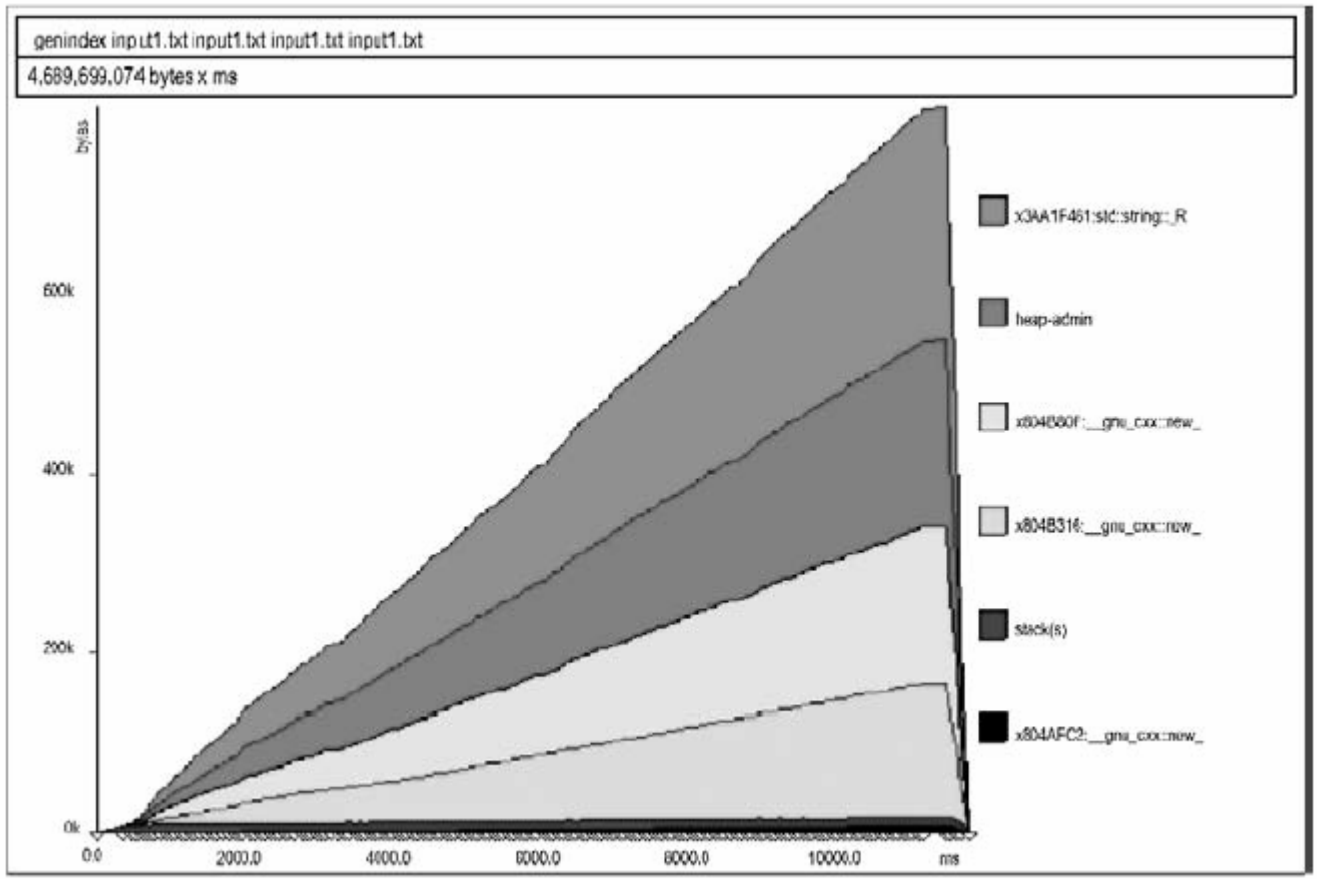
Bug fix

```
119 while(1) {
120 c = getc(fp);
121
122 if(c == EOF || c == ’\n’) {
123 add_to_index(newword, current_line);
124 newword = "";
125 current_line++;
126 lines.push_back(buffer);
...
128 buffer = ""; // <---- this is the bug fix
```

Memory usage after the fix

```
> g++ -g -o genindex -DFIX_LINES genindex.cc
> ./genindex input1.txt input1.txt input1.txt input1.txt >log
-- memory size for index of 'input1.txt' file size=20016:
--     filename=14 wordindex=47211 lines=19318 total=66543
...
-- memory size for all data structures: 266172 bytes
```

Massif output



Questions?