Project 3 Design Document

Purple Team
Ian Ross, Sergio Oritz, Alex Sadeghi, Erik Ziegenbalg, Jesse Cheun

1 Design

In this document we will specify the implementations of slug_malloc and slug_free, functions that will wrap normal malloc() and free() detecting common allocation problems.

1.1 Binary tree data structure

In order to keep track of memory allocations, it was necessary to create a data structure which would hold all the information of each call to slug_malloc. We decided to use a binary tree, sorted by address, to store the information. Each node in the tree would hold: the address returned by malloc(), the size of the allocation, the filename and line number of the call to malloc() and the timestamp of the call.

We decided to use a binary tree rather than a linked list for increased performance when a large number of allocations occur. Rather than using an array to implement our binary tree, we created a struct in tree.h along with all tree operations in order to add modularity to our design.

1.2 Slug_malloc

This function wraps the normal malloc function that checks for additional errors. Slug_malloc will terminate the program when trying to allocated more than 128MiB. It will print out an error it is trying to malloc a size of zero.

1.21 Getting the timestamp

Using the sys/time library, the gettimeofday() function is use to get the system current time. On a successful malloc the timestamp is recorded and stored in the tree data structure.

1.22 atexit() function call

On the first successful call to slug_malloc() an custom exit handler is set to slug_memstat() which prints out any unfree addresses and summaries of current and all allocations. A global flag exit_handler is use to check if the handler has been installed to prevent multiple exit handlers to be set.

1.23 Inserting an allocation to the tree

When slug_malloc() finish allocating memory a new node is created using create_node() storing the address pointer, size of the allocation, where slug_malloc() was called, and the timestamp. The new node is then inserted into the tree using the insert() function from the tree library.

1.3 Slug_free

This function is intended to wrap the normal free() function in order to check for common errors such as freeing an invalid address, or freeing an address that has already been freed. The function also needs to alert the data structure that an address has been freed so it won't print the address when the program exits.

1.31 Check to see if address is valid

In order to free an address we must first check to see if the address is valid, this means checking to see if the address is in the tree. Here we call the isInTree() function which takes an address and recursively searches the tree for that address. If the address is found the node containing that address in the tree is returned and if not NULL is returned.

1.31 Has the address already been freed

Next, we check if the freed flag is set to true and if it is, the address has already been freed and an error message should be printed. After this error is printed, as per the specifications, the program exits and dumps the rest of the tree that has yet to be freed showing the user a summary of their memory leaks.

1.32 Free the address

If the address is indeed valid, it now needs to be freed using the normal free() function. After this, the freed flag is set to true indicating to the data structure that the memory of a particular node has been freed.

1.33 Invalid address

If the address is not in the tree at all, it has not been allocated and thus an error message must be printed and the program is terminated and once again the program dumps the tree to show the user any memory leaks.

1.4 slug_memstat

This function adds extra functionality to the normal memstat function. Users can call this function at normal program exit using the command: atexit(slug_memstat(void))

1.41 Printing the tree

The tree holds fields for size, timestamp, address of location, and file and line number in test program where the allocation occurred. By calling “print_tree(root)”, we are able to get a neatly formatted summary of all these important memory statistics.

1.42 Allocations/Memory

The variables used for holding information about allocations and memory were held inside a struct defined in purple_malloc.h called mem_info. We kept track of allocations and memory size by simply incrementing a counter every time slug_malloc() was called. To keep track of active allocations and memory size we simply decremented these counters in slug_free().

1.42 Mean calculation

The mean was very straightforward: If no allocations were made, a mean of 0.0 returned. In all other cases the total size of all the memory allocated was divided by the total number of allocations and then returned.

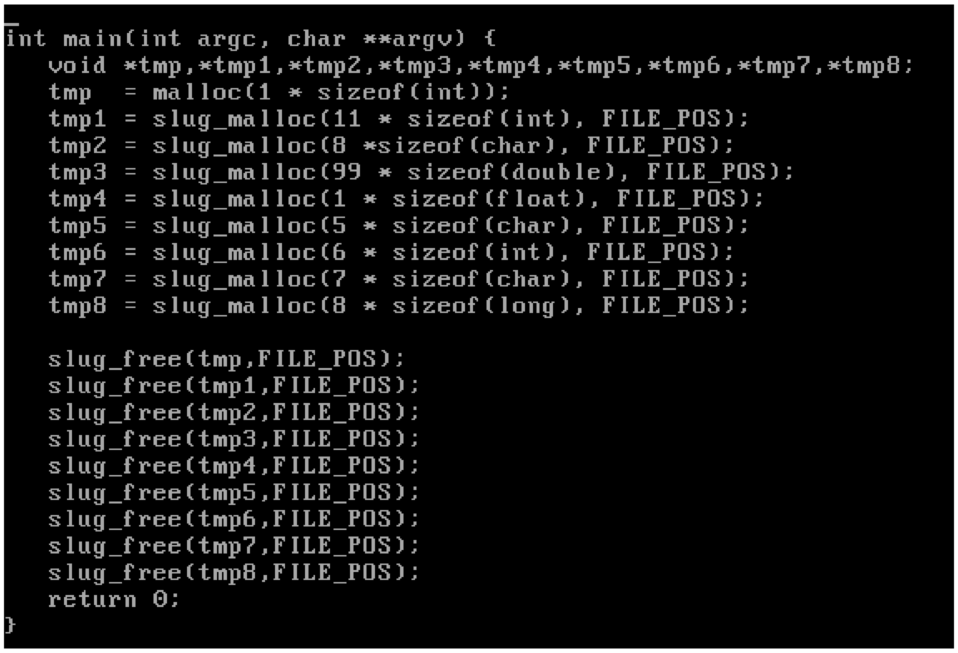
1.43 Standard deviation calculation

For standard deviation, we first solved the problem of how to keep track of running standard deviation by creating two empty arrays of max length 100, one to keep track of (all sizes allocated(lens). And another to keep track of (lens - mean)^2. (tmp). After that we followed the required steps to obtain standard deviation. Note: our formula computes the Population Standard Deviation.

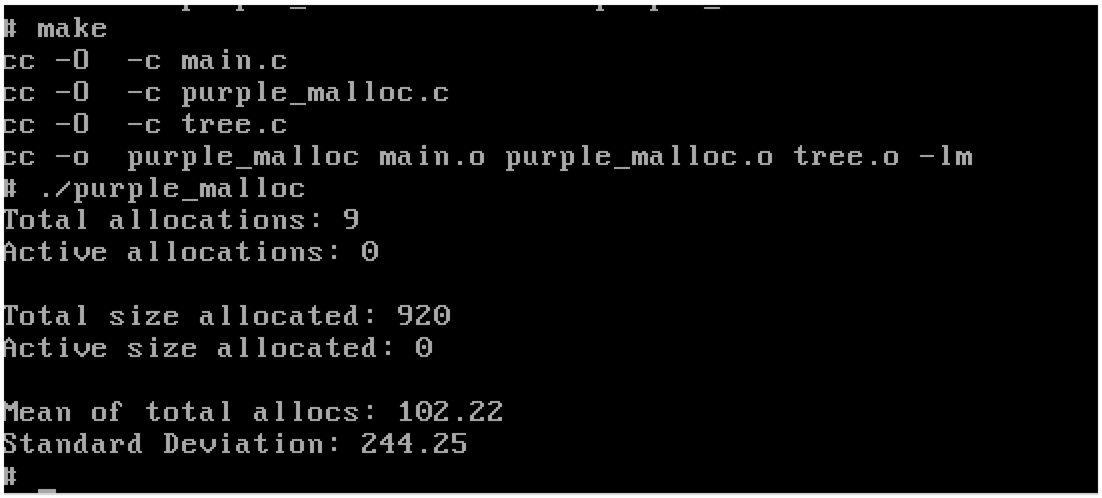
2 Testing

In order to test our program we needed to run a slightly different test for each specification to ensure it was working.

2.1 Regular allocation and deallocation behave correctly

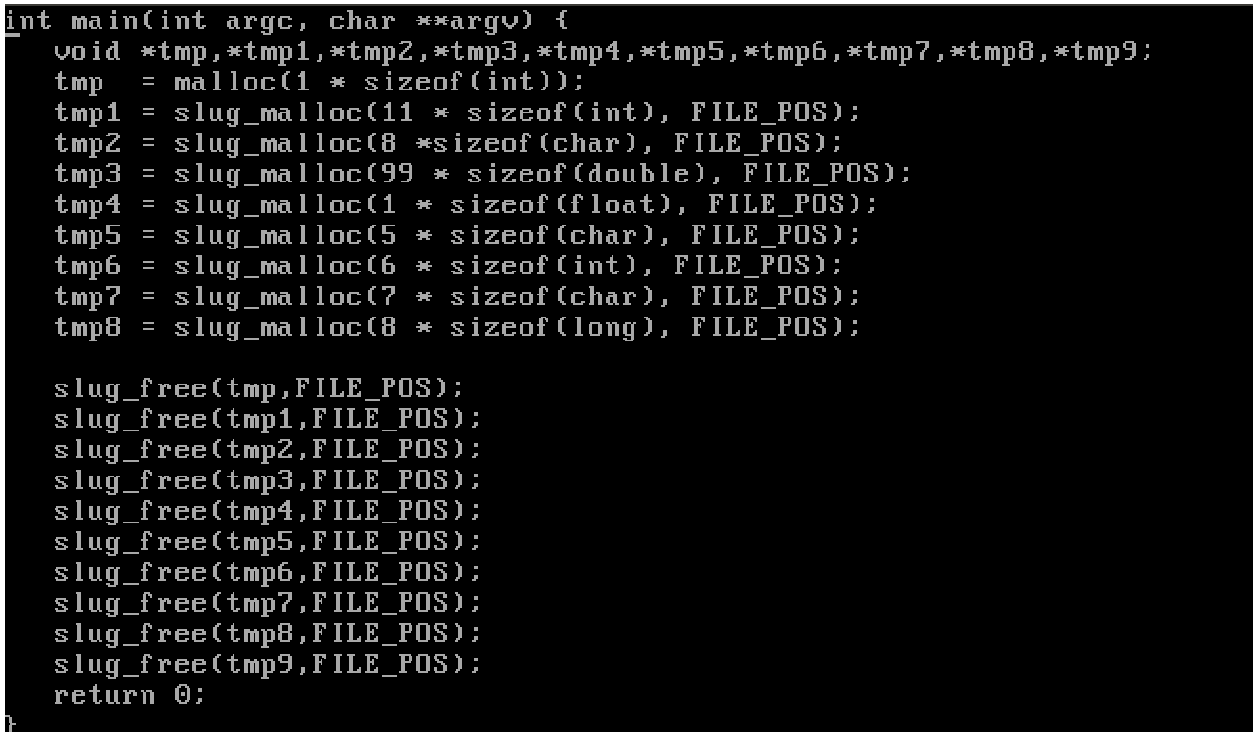


```
int main(int argc, char **argv) {
   void *tmp,*tmp1,*tmp2,*tmp3,*tmp4,*tmp5,*tmp6,*tmp7,*tmp8;
   tmp  = malloc(1 * sizeof(int));
   tmp1 = slug_malloc(11 * sizeof(int), FILE_POS);
   tmp2 = slug_malloc(8 *sizeof(char), FILE_POS);
   tmp3 = slug_malloc(99 * sizeof(double), FILE_POS);
   tmp4 = slug_malloc(1 * sizeof(float), FILE_POS);
   tmp5 = slug_malloc(5 * sizeof(char), FILE_POS);
   tmp6 = slug_malloc(6 * sizeof(int), FILE_POS);
   tmp7 = slug_malloc(7 * sizeof(char), FILE_POS);
   tmp8 = slug_malloc(8 * sizeof(long), FILE_POS);

   slug_free(tmp,FILE_POS);
   slug_free(tmp1,FILE_POS);
   slug_free(tmp2,FILE_POS);
   slug_free(tmp3,FILE_POS);
   slug_free(tmp4,FILE_POS);
   slug_free(tmp5,FILE_POS);
   slug_free(tmp6,FILE_POS);
   slug_free(tmp7,FILE_POS);
   slug_free(tmp8,FILE_POS);
   return 0;
}
```

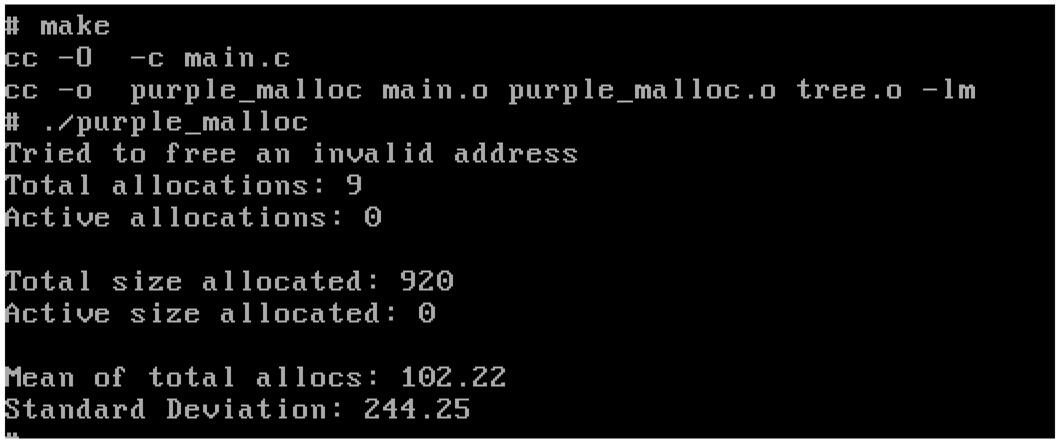


```
# make
cc -O  -c main.c
cc -O  -c purple_malloc.c
cc -O  -c tree.c
cc -o  purple_malloc main.o purple_malloc.o tree.o -lm
# ./purple_malloc
Total allocations: 9
Active allocations: 0

Total size allocated: 920
Active size allocated: 0

Mean of total allocs: 102.22
Standard Deviation: 244.25
#
```

2.2 Deallocating an invalid address immdiately detected



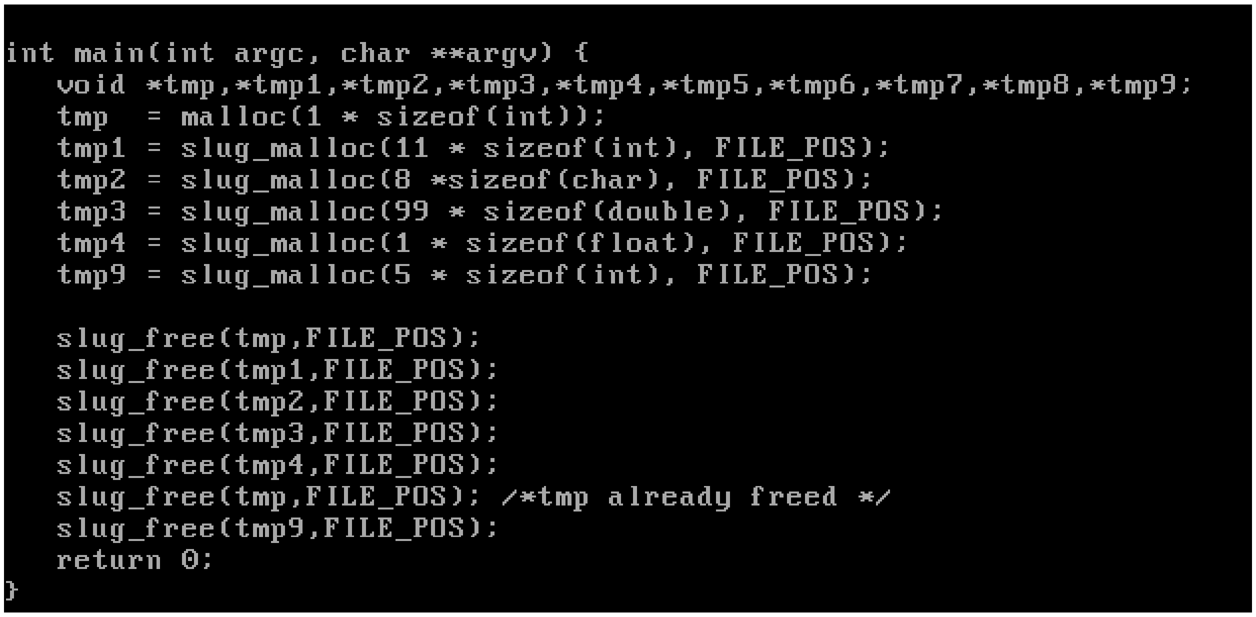
```
int main(int argc, char **argv) {
   void *tmp,*tmp1,*tmp2,*tmp3,*tmp4,*tmp5,*tmp6,*tmp7,*tmp8,*tmp9;
   tmp  = malloc(1 * sizeof(int));
   tmp1 = slug_malloc(11 * sizeof(int), FILE_POS);
   tmp2 = slug_malloc(8 *sizeof(char), FILE_POS);
   tmp3 = slug_malloc(99 * sizeof(double), FILE_POS);
   tmp4 = slug_malloc(1 * sizeof(float), FILE_POS);
   tmp5 = slug_malloc(5 * sizeof(char), FILE_POS);
   tmp6 = slug_malloc(6 * sizeof(int), FILE_POS);
   tmp7 = slug_malloc(7 * sizeof(char), FILE_POS);
   tmp8 = slug_malloc(8 * sizeof(long), FILE_POS);

   slug_free(tmp,FILE_POS);
   slug_free(tmp1,FILE_POS);
   slug_free(tmp2,FILE_POS);
   slug_free(tmp3,FILE_POS);
   slug_free(tmp4,FILE_POS);
   slug_free(tmp5,FILE_POS);
   slug_free(tmp6,FILE_POS);
   slug_free(tmp7,FILE_POS);
   slug_free(tmp8,FILE_POS);
   slug_free(tmp9,FILE_POS);
   return 0;
}
```

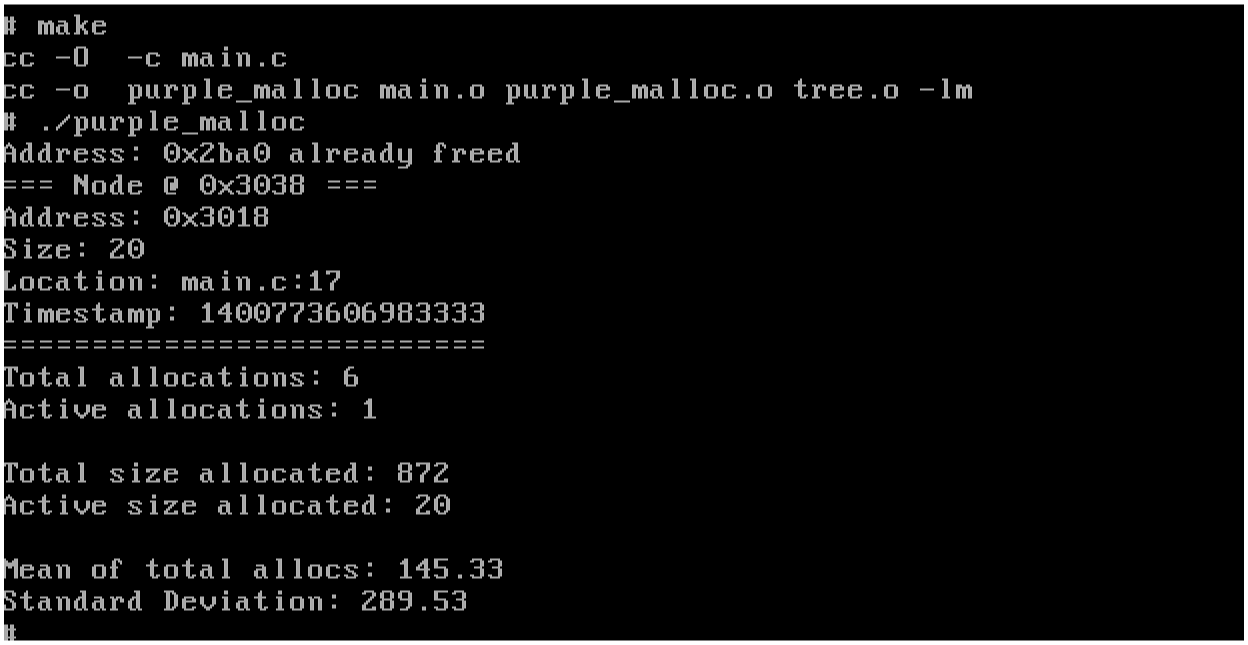


```
# make
cc -O  -c main.c
cc -o  purple_malloc main.o purple_malloc.o tree.o -lm
# ./purple_malloc
Tried to free an invalid address
Total allocations: 9
Active allocations: 0

Total size allocated: 920
Active size allocated: 0

Mean of total allocs: 102.22
Standard Deviation: 244.25
#
```

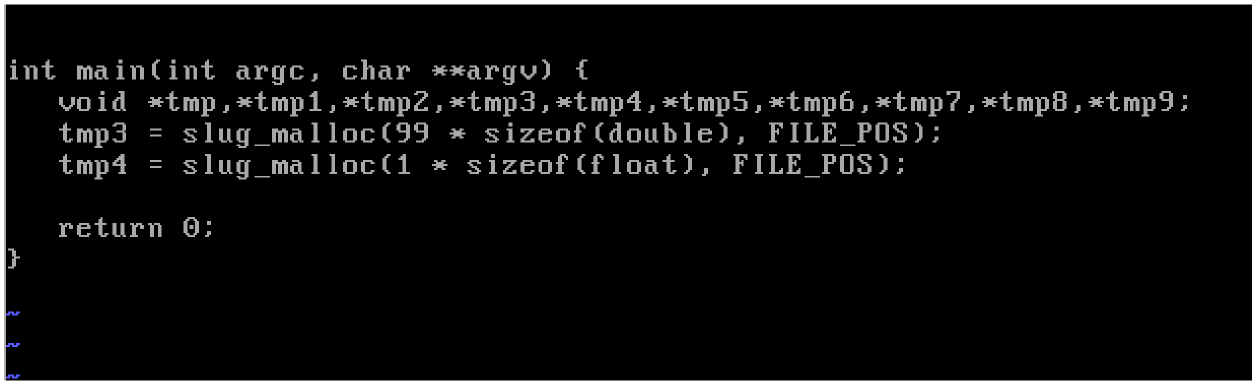
2.3 Deallocating already freed address immediately detected

Address that has already been freed is printed and then the tree is dumped showing only one node remaining that needs to be freed

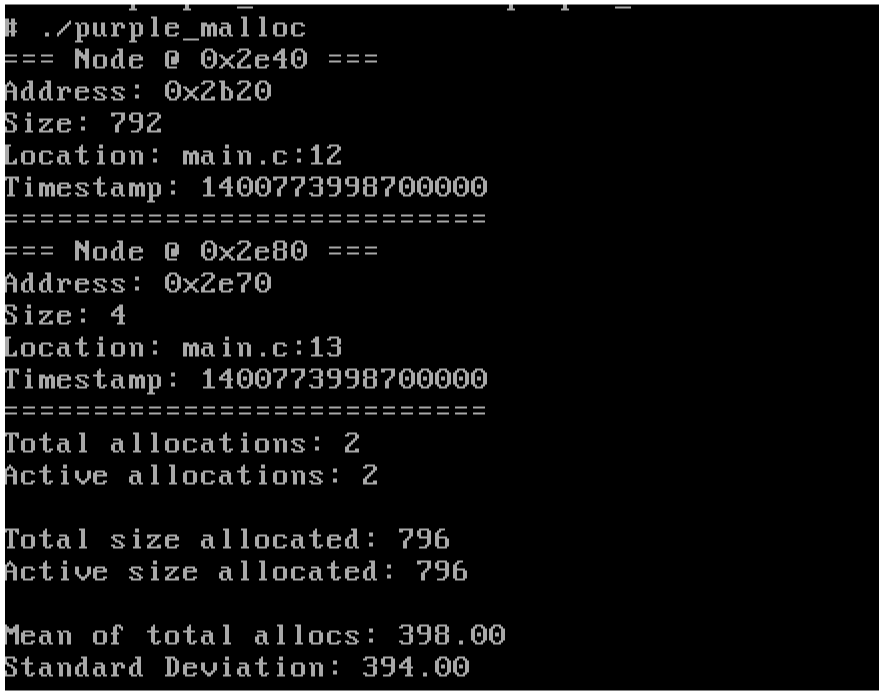

```
int main(int argc, char **argv) {
   void *tmp,*tmp1,*tmp2,*tmp3,*tmp4,*tmp5,*tmp6,*tmp7,*tmp8,*tmp9;
   tmp  = malloc(1 * sizeof(int));
   tmp1 = slug_malloc(11 * sizeof(int), FILE_POS);
   tmp2 = slug_malloc(8 *sizeof(char), FILE_POS);
   tmp3 = slug_malloc(99 * sizeof(double), FILE_POS);
   tmp4 = slug_malloc(1 * sizeof(float), FILE_POS);
   tmp9 = slug_malloc(5 * sizeof(int), FILE_POS);

   slug_free(tmp,FILE_POS);
   slug_free(tmp1,FILE_POS);
   slug_free(tmp2,FILE_POS);
   slug_free(tmp3,FILE_POS);
   slug_free(tmp4,FILE_POS);
   slug_free(tmp,FILE_POS); /*tmp already freed */
   slug_free(tmp9,FILE_POS);
   return 0;
}
```


```
# make
cc -O  -c main.c
cc -o  purple_malloc main.o purple_malloc.o tree.o -lm
# ./purple_malloc
Address: 0x2ba0 already freed
=== Node @ 0x3038 ===
Address: 0x3018
Size: 20
Location: main.c:17
Timestamp: 1400773606983333
===========================
Total allocations: 6
Active allocations: 1

Total size allocated: 872
Active size allocated: 20

Mean of total allocs: 145.33
Standard Deviation: 289.53
#
```

2.4 Allocating memory and then exiting triggers the leak detector



```
int main(int argc, char **argv) {
   void *tmp,*tmp1,*tmp2,*tmp3,*tmp4,*tmp5,*tmp6,*tmp7,*tmp8,*tmp9;
   tmp3 = slug_malloc(99 * sizeof(double), FILE_POS);
   tmp4 = slug_malloc(1 * sizeof(float), FILE_POS);

   return 0;
}

~
~
~
```



```
# ./purple_malloc
=== Node @ 0x2e40 ===
Address: 0x2b20
Size: 792
Location: main.c:12
Timestamp: 1400773998700000
===========================
=== Node @ 0x2e80 ===
Address: 0x2e70
Size: 4
Location: main.c:13
Timestamp: 1400773998700000
===========================
Total allocations: 2
Active allocations: 2

Total size allocated: 796
Active size allocated: 796

Mean of total allocs: 398.00
Standard Deviation: 394.00
```