Allocation tracker

The aim of this project is to develop a programming aid that will enable memory management to be analysed in complete (possibly modular, but executable) code.

This library will act as a patch at compile time, diverting memory management functions to clones that can be used to gather information at runtime.

General presentation

Alternative/complement to Valgrind

Valgrind is a very powerful tool - sometimes too powerful - for detecting memory problems.

Its main drawback is that it acts directly on the .out *executable*: it therefore analyses everything the process does, i.e. our code but also all calls to external libraries whose source code is not available (and therefore cannot be corrected).

You only need to use Valgrind on a very simple program using a graphics API (OpenGl, SDL or other) to realise this: lots of 'problems' are detected even if our code doesn't manage memory directly.

The tool proposed here does not act directly on the executable but upstream, at compilation time, by diverting certain functions (in this case, those managing memory) to 'clones' that will collect information, check certain points and try to overcome the errors detected.

As with the gdb debugger, this tool is activated **without any intervention in the source code**, but requires a **parameterised compilation** (cf. the -g option for 'plugging in' gdb).

Features

The analysis will focus at least on allocation/release (malloc, calloc, realloc, free) - in <stdlib.h> - but may be extended to memory block manipulation functions (memset, memcpy, memmove) - in <string.h>.

Like any *debugging* "option", this library is intended to be used in a project development/correction phase. As such, it must be possible to activate/deactivate it at compile time (on the command line or via a Makefile).

Its use must not modify the behaviour of the target program, except possibly to overcome a few known bugs and faults in some of these functions (see example 2 below).

Its role will be to :

- ▹ identify and count calls to memory allocation / manipulation functions,
- ▹ evaluate the amount of memory allocated / released globally,
- ▹ check the corresponding releases,
- ▹ spot and overcome a few classic *bugs*.
- ▹ test a few possible overflows
- ▹ produce a final report

Some illustrative examples

The following short program allocates a 1-byte block, displays the address and the pointed value, but does not free the allocated area. It compiles without errors or warnings and runs without problems, but there is no indication that the memory has not been freed.

utest_malloc.c

```c
# include <stdio.h>
# include <stdlib.h>

int main(void)
{
  char* ptr=(char*)malloc(1);
  fprintf(stderr, "address: %p -> value: %d",ptr,*ptr);
  return 0;
}
```

xterm

```
$> gcc -Wall utest_malloc.c -o
utest_malloc
$> ./utest_malloc
   address:0x55e852da42a0 -> value:0
$>
```

☞ with a slight adaptation of the compilation directive (detailed below), running this same program (still compiled without errors or warnings) will produce something like :

xterm

```
$> ./utest_malloc
  in file <utest_malloc.c>
    in function <main>
      at line <006> : 0x5581bf72b2a0=malloc(1)
  address:0x55e852da42a0 -> value:0
    summary :
      @0x5581bf72b2a0 : 1 byte allocated (src/utest_malloc.c|main|6) not freed
```

Another example:

This other program lines up several 'wrong' or 'badly done' instructions (lines `08,09,11`) but still compiles without error or warning. This time, however, execution goes very badly.

utest_free.c

```c
01  # include <stdio.h>
02  # include <stdlib.h>
03
04  int main(void)
05  {
06    int* ptr1=malloc(1);
07    int** ptr2=&ptr1;
08    free(*ptr2);
09    free(ptr1);
10    *ptr2=NULL;
11    free(ptr2);
12    return 0;
13  }
```

xterm

```
$> gcc -Wall utest_free.c -o utest_free
$> ./utest_free
  free(): double free detected in tcache
  2 Abort (core dumped)
$>
```

Here again, the use of our library will help to identify problems and avoid execution errors:

xterm

```
$> ./utest_free
    - (libmtrack) automatic activation> stderr - ------------
  -----------------
  Thu Jul 11 12:07:01 2024
  USER: me
  HOST : d3ll-5530
  DIR. : /home/machin/truc/Code/ ----------------------------
  --
  line 006, function <main> of <utest_free.c> malloc(1)->0x559bd6078750 line
  008, function <main> of <utest_free.c> free(0x559bd6078750)
  line 009, function <main> of <utest_free.c> free(0x559bd6078750) - ERROR: "double free" detected line
  011, function <main> of <utest_free.c> free(0x7fffb8e7b118) - ERROR: illegal input -------------------
  -----------
  balance sheet
  malloc: 1 call
  free   1 correct call
  free   2 incorrect calls (ignored)
  total allocates 1 byte
  total freed 1 byte (100.0%)
```

The format and content of the information displayed is largely adaptable and nothing is imposed as long as the expected functionalities are present and the information given is relevant.

Another version of this library, on the same program, could give :

xterm

```
$> ./utest_free
  in file <utest_free.c>
    in function <main>
      at line <006> : 0x56053af6c2a0=malloc(1)
      at line <008> : free(0x56053af6c2a0) ->
      OK
      at line <009> : free(0x56053af6c2a0) -> error : address already freed (src/utest_free.c|08|main)→ ignored
      at line <011> : free(0x7ffdeb00eef8) -> error : address not listed→ ignored
    summary :
      0x5581bf72b2a0 : 1 byte allocated (src/utest_min.c|main|7) -> freed(src/utest_free.c|main|08)
```

These two versions, which are already quite sophisticated, also announce the file and function names, as well as the line number where the call is being made. They are also capable of detecting and overcoming certain "false manoeuvres".

Specifications

The aim is to develop a "transparent" shared utility library. As a result, while you are entirely responsible for the development choices you make, the *deliverable* must comply with a fairly rigorous set of specifications to ensure that the library functions properly.

Packaging

- interface: accessible header files (*.h) and shared library (lib*.so)
- connection: automatic via Makefile and/or shell command (direct compilation).
- installation: by shell script
- scope of analysisall dependencies compiled and assembled
- analysis : standard output and/or text file (no specific format).
- unit tests: a set of short test programs illustrate functionality.
- README: a file explaining how to use your lib (installation, compilation, execution, tests).

Features

- statistics: count calls to memory management functions, evaluate the amount of memory allocated and released, the recycling rate of allocation blocks, their average size, etc.
- review: review previous statistics, detect any memory blocks that have not been released and indicate their location
- debug: systematically identify and report classic failures and *bugs* linked to the use of the free function: release on a null, illegal or already released address ("double free corruption") and ensure that the program can overcome these problems (avoid seg. fault).
- *verbose* mode: trace all calls, indicating the call file, call function, line number and :
  - for allocation functions: block size, return address, recycling rate
  - for the release function: the location of the allocation call, reporting any anomalies (allocation and release in different files, for example).
  - for block movement functions: the sizes and addresses of the source and destination blocks.
- output: by default the output (displays) will be the standard error file (`stderr`) but can be redirected to a text file stored in memory.

  In this case, some additional information should be included:
  - host program creation (compilation date)

- completion
- user name (USER) and machine name (HOST)
- condition use: host program with list arguments
- plotter parameters
- ... and anything else you deem useful ...

Implementation

Key functions

The easiest part of this project is rewriting the cloned functions. At this level, the only difficulty lies in the data structure to be adopted to store the information collected: it is impossible to predict the quantity of data that will need to be stored (imagine the extreme case of allocation in an infinite loop... or test your previous projects...). A 'protection' mechanism will therefore be needed to deal with this kind of situation.

So you need a data structure that is flexible enough to grow easily without multiplying allocations (that would be counter-productive!!!) and a limit beyond which the plotter deactivates itself (if it has to come to that, it's probably because the host program needs to be revised!)

> Of course, the plotter must not be self-analysing: if it uses memory manipulation functions, these **must not** be replaced by their clones at compile time.

The free and realloc functions require particular attention because of their design flaws (source of many Seg.Fault errors):

- `free(ptr);` does not check the validity of the address passed as a parameter.
- `free(ptr);` frees the area pointed to by ptr but keeps the address (now invalid) in ptr,
- `dst=realloc(src,size);` does not check the validity of the address contained in src and if dst≠ src, the src pointer contains an invalid address (same fault as free).

The scope of data, variables, functions, etc.

This is of course a crucial point in this project:

- the tracer must be able to extract as much information as possible from the host program, but must under no circumstances interfere with its operation (except to overcome certain problems and avoid errors).
- its operation must be transparent and its architecture invisible from the outside, and it must be easy to configure.

Unit testing

A unit test generally involves setting up the simplest possible execution environment to test a specific functionality of the program.

Since the purpose of the library developed here is to analyse the execution of another program, these tests will obviously have to be small programs that can be compiled and executed.

For example, the minimal test code given on p.1 can be used as a unit test for :

- automatic plotter activation
- tracing the malloc function
- detection of unreleased blocks.

Environment / Installation

This part goes a little beyond the scope of simple C programming and will require a few additional operations such as writing the Makefile and defining the appropriate environment variables and compilation commands (configuration file~ /.bashrc).