A practical guide to computer simulation II

Alexander K. Hartmann, University of Göttingen

May 14, 2003

3.3 Memory checker

Memory faults, especially for dynamically allocated memory: read of unitilialized memory, read/write past array boundaries, access to already freed memory, usage of wrong pointers, not-freeing of not used memory (program will grow in size).

Memory faults are hard to detect: bug becomes visible (if at all) usually at totally different place. Example, see Fig. 1:

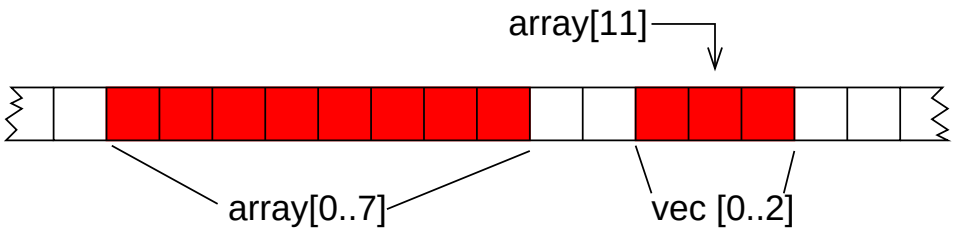

Figure 1: Two dynamically allocated arrays. Writing beyond the limit of the first one, e.g. into `array[11]`, destroys the content of the second array.

General tool: Purify. But commercial program (4000 Euro). For free: Checkergcc, but outdatet (works only with old gcc). Here:

Electric fence: Detects access past array boundaries (dynamically allocated) and access in freed memory. Test program `fencetest.c`:

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

int main()
{
    int t, *array, sum = 0;

    array = (int *) malloc (100*sizeof(int));
    array[50] = 1;
    array[200] = 1;              /* writes past boundaries */
    free(array);
    array[50] = 1;              /* writes in freed memory */
    return(0);
}
```

Compile

```
gcc -o fencetest fencetest.c -g -lefence
```

Run with `gdb`:

```
gdb fencetest
(gdb) r
  Electric Fence 2.2.0 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>

Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 1024 (LWP 2136)]
0x080485cc in main (argc=1, argv=0xbffff094) at fencetest.c:10
10          array[200] = 1;              /* writes past boundaries */
(gdb)
```

Basic principle of `efence`: memory is allocated such that the allocated range ends just in front of a "forbidden" memory range. When accesing past the boundary a segmentation violation occurs.

`efence` detects also access to freed memory: Remove line 10, compile + run again.

To test access *before* allocated region:

```
export EF_PROTECT_BELOW=1
```

change line 10 to

```
    array[-200] = 1;              /* writes before boundaries */
```

More elaborated: `checkergcc`, but works only for old gcc (2.8.1).

4 Software Engineering

Refs. [1, 2]

In the 1970s: "software crisis": Large programs were hard to finish, many problems. → Use methods from engineering.
First think and plan, then write program!!! Saves later much time, if program longer than 100 lines.

4.1 Software production

Sample software creation process (one person or few people)

1. Definition of problem, solution strategies
   - Draw diagrams.
   - Discuss with others.
   - Top down approach

- What is the input? How to pass?
- What is output? How to store, analyze?
- Foresee extensions.
- Code re-usage (own/libraries).
- Problem solvable? Which algorithms?

2. Design data structures.

   - Often one wins a lot speed by introducing new (redundant) data structures.

     Example: Array of atom data plus list of events. If event of belonging to an atom has to be removed: one must run through full event list. Better include pointers to list elements in ataom data → direct access.
   - Use existing structures classes?

3. Design small tasks:

   - Bottom up approach.
   - Flexibility.
   - Iteration Design, Implementation, test, further design etc ("software cycle").

4. Distributing work

   - Each worker one module
   - Hide implementation, clear interfaces.
   - Use source-code management system (RCS).

5. Implementation. Style: see below. One file per module. Use *Makefile*.

6. Testing

   - First test single modules.
   - Read code again.
   - Test immediately.
   - Use `-Wall` option
   - Use `lint`.
   - Find special or borderline cases.
   - Use debugger.
   - Check memory management.
   - In case of hard to find bugs: stop after two hours, often you will get an idea for the solution
   - Never write programs if you are tired (e.g. not after 21.00 h).
   - Restrict scopes of variables/data types/value ranges.
   - Provide subroutines for printing instances.
   - Test systems with know results.

- Test on small systems.
- Apply different ways to get the same result. (E.g. check old specialized against new generalized code).
- Compare with bounds, approximations.
- Check for invariants (energy, momentum).
- Later do integration test for several modules.
- Keep old (working) versions.

7. Documentation

   - Do it! Six months later you will benefit.
   - Now!
   - Comments in code.
   - Online help (`-help` option)
   - External documentation (`man` pages) : if more than one user.

8. Using

   - Estimate duration of runs.
   - Have a good statistics (see error calculation later on)
   - Store results: many low-level data allows maybe another evaluation of the data. But beware of too much memory usage.
   - Write informations about how data was generated

4.2 Programming style

Ref. [3].

- Use a personal but consistent style.
- Split code into meaningful modules (easier compiling, testing, reusing, distributing of work)
- Use separate header files for data structures.
- Use meaningful names for variables and functions.
- Use proper indentation.
- Usually one command per line.
- Don't use goto style.
- Do ** **NOT** ** use global variables: only faster in the beginning. Later: less flexibility, more bugs, more effort.
- Use comments:
  - Module comments, including version history
  - Type/data structure comments

- Subroutine comments: explain use and parameters (in/out), preconditions, references
- Block comments (25 lines)
- Line comments, if not obvious meaning

4.3 Object oriented programming

Object-oriented (OO) languages: C++, Smalltalk, Eiffel. But OO design also possible *without* an OO language, i.e. with C, Pascal or Fortran. Be flexible, but not orthodox.
Basics ([4]):

- *Objects* fall into *classes*, characterized by values (=states) and operations (methods).
- Data capsuling: Hide actual implementation (black box). Facilitates reuse and changes.
- Inheritance: Top down: Lower level class = special case of higher level class. Bottom up: Assemble higher level object from lower level objects.
- Function/Operator overloading: Define + Operation for graphs: makes code more readable and programming more convenient.

References

[1] I. Sommerville, *Software Engineering*, (Addison-Wesley, Reading (MA) 1989)

[2] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*, (Prentice Hall, London 1991)

[3] B.W. Kernighan and R. Pike, *The Practice of Programming*, (Addison-Wesley, Reading (MA) 1999)

[4] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, (Prentice Hall, London 1991)