Static Code Analysis

(without paying too much)

Oleg Goldshmidt

olegg@voltaire.com

How Do You Spend Your Time?

- in a typical SW development project — in the debugger
  - especially if you don't invest in requirements, specifications, design, reviews, error handling, logging, testing, and — in general — thinking...
- debugging is the worst possible way to waste time
  - requires immense patience and concentration
  - often very difficult (e.g., concurrency)
  - most people find it uninteresting and hate it
  - unproductive:
    "I'm afraid that I've seen too many people fix bugs by looking at debugger output, and that almost inevitably leads to fixing the symptoms rather than the underlying problems." [Linus Torvalds]

Finding Bugs In Code

- we have a great QA department, we have bug-related release criteria, we have procedures...
- “It takes three times the effort to find and fix bugs in a systems test than when done by the developer. It takes ten times the effort to find and fix bugs in the field than when done in system test.” [Larry Bernstein]
- “Testing can show the presence of bugs, not their absence.” [Edsger W. Dijkstra]
- if we want bug-free code testing is essential, but not nearly enough
- the important skill is not to find bugs, but avoid them in the first place

There Are No Trivial Bugs

- most bugs are symptoms of a design problem
  - you will not figure it out by stepping through
- must pay attention even to the simplest things - they are symptoms of larger problems
- real production code:

```
class Object {
    public:
        long shadow;
        static void Cast(long shadow);
};

void Object::Cast(long shadow) {
    shadow = 1L;
}
```

Paying Attention

- the real issues belong to the realm of design
- design requires understanding
- but what about the “symptoms”?
  - very difficult to notice by the “naked eye”
  - there are (relatively) simple rules, but they are numerous, and appling them is too tedious
  - computers are much better than humans in applying numerous simple rules automatically
- there are various code analysis tools that can flush the symptoms for you
- they will not understand your design intent, so they’ll flag the problems, but won’t fix them

Know Thy Compiler

- the readily available static analysis tool
- modern compilers will warn you about many different symptoms of deeper problems
- you won't win the lottery without buying a ticket
- ask the compiler to warn you, and it will
- do not ignore the warnings!
  - do not treat them like a nuisance
  - do not fix the symptom
  - investigate in depth: is there a design problem the compiler is trying to warn you about?
    - remember: the compiler does not understand design intent
  - fix the design — avoid numerous other bugs

Portability

- platform: CPU arch, OS, compiler, libraries, linker
  - assume 3 of each: $3^5 = 243$ platforms easily
- practically impossible to test all combinations
  - do you have access to all?
  - do you have enough HW?
  - do you have enough time?
  - will your code still work if your employer or your client decides to deploy another platform?
- yes, it is possible to develop portable code
- "Who cares? My code only needs to run on Windows!"
  - war story: port to Linux in 2 months or you are fired!
  - what about the next version of MSVS / `cl.exe` ?

Portability (Cont.)

- standards: POSIX (and Windows), ISO/ANSI, XOPEN, SUS, etc.
- compilers are supposed to implement language standards, but normally only approximate them
  - some are better than others
  - may implement many extensions, but warn about non-standard constructs
  - choose a portable and compliant compiler
    - but don't forget to test on important target platforms
- same goes for linkers, though they are more esoteric
- treat the build system as part of the code
  - and make it portable!

Thinking in the Future Tense

- a header file from real production code

```
// [name withheld to protect the guilty - OG]
// Apr-09-2003
// I removed the non-Win32 leg since the
// pthread_cond_wait code did not always work
// (needed to be waiting when signal set or
// signal was lost). Since all platforms now
// either are Win32 or have Win32 emulation
// layer, just use the Windows API's.
```

- guess what: the code needed to be ported to a platform without a Win32 emulation layer, and there was no version control...

GCC Examples

- very portable compiler
- implements many extensions, but can be made very strict
- compile new code with `-D_POSIX_SOURCE`, `-D_ISOC99_SOURCE`, etc.
  - or put the `#define`'s into your code template
  - RTFM, RTF `/usr/include/features.h`
- use `-pedantic` for strict conformance
  - `-ansi`, `-traditional` if you have very old code
- aside: for Windows, use MSDN documentation, it is much better now than what it used to be

Selected GCC Warnings I

```
'-Wchar-subscripts'
  Warn if an array subscript has type
  'char' (may be signed)
'-Wformat'
  Check calls to 'printf' and 'scanf',
  etc., to make sure that the arguments
  supplied have types appropriate to
  the format string specified (related
  options, e.g., -Wformat-security)
'-Wunused'
  Warn whenever a static function,
  argument, label, local variable, or
  computed result is declared but not used
'-Wfloat-equal'
  Don't compare FP numbers for equality!
```

Selected GCC Warnings II

'-Wparentheses'
Do you remember the operator precedence table?

What is x<=y<=z? Are you sure?

Is this what the programmer meant?

```
{
  if (a)
    if (b)
      foo();
  else
    bar();
}
```

Selected GCC Warnings III

Falling through `switch` :

```
enum {
   FIRST_CASE=1,
   SECOND_CASE,
   THIRD_CASE
} c;
int n;
...
switch (c) {
case FIRST_CASE: n = 1; break;
case SECOND_CASE: n = 2; break;
}
do_something_with(n);
```

Use `-Wswitch`, `-Wswitch-default`, `-Wswitch-enum`

Selected GCC Warnings IV

```
'-Wsequence-point'
  C standard defines a partial ordering in term
  of "sequence points". Examples: after
  evaluation of an expression, after first
  operand of a &&, ||, ?: or 'comma', before
  function call (but after evaluation of the
  arguments), etc. Only a partial order (no
  order between function arguments, between
  function calls, etc.) is defined.

  Can catch undefined behavior:
  a = a++;
  a[n] = b[n++];
  a[i++] = a[i];
  x = func(a[n], b[n++]);
```

Selected GCC Warnings V

`'-Wuninitialized'`

- Obvious? Even clobbering with `setjmp` ?
- Need to understand the compiler: works only with `-O`
- Related: `-Winit-self`
- Works for `struct` 's, does not work for `volatile` 's
- Optional because of false positives:

```
switch (y) {
case 1: x=1; break;
case 2: x=4; break;
case 3: x=5; break;
}
foo (x);
```

```
int save_y;
if (change_y)
  save_y=y, y=new_y;
...
if (change_y)
  y=save_y;
```

GCC Warning Aggregations

- `-Wall` : warns about questionable constructs that can be easily avoided.
  - does not include ALL warnings!
  - does not include some variants of included warnings
  - does not warn about structures that one may want to check for, but are sometimes necessary or hard to avoid
- `-Wextra` (used to be `-W`): a long list of warnings of different kinds
  - does not include all warnings, either

Some Unaggregated GCC Warnings

```
'-Wshadow'
'-Wpointer-arith'
  Warn about anything that depends on the
  "size of" a function type or of 'void'.
'-Wcast-qual'
  Warn whenever a pointer is cast so as to
  remove a type qualifier from the target
  type. For example, warn if a 'const char*'
  is cast to an ordinary 'char*'.
'-Waggregate-return'
  Warn if any functions that return structures
  or unions are defined or called.
'-Wunreachable-code'
  Warn if the compiler detects code that
  will never be executed.
```

The Most Important GCC Option

```
'-Werror'
  Convert all warnings into errors.
```

Static Analysis Beyond Compiler

- multiple compilers
  - different capabilities
  - improved portability
- lints
- commercial static analysis tools
- dynamic analysis
- memory checkers

Lints

- originally a specific tool, now any static analysis tool that flags suspicious or non-portable constructs
- PC-lint and FlexeLint from `http://www.gimpel.com`
- many different open source implementations
- specialized lints
  - C++ — `cpplint` : `http://sourceforge.net/projects/cpplint/`
  - security — `splint` `http://sourceforge.net/projects/splint/`
- PC-lint output analyzer `http://sourceforge.net/projects/aloa-lint/`
- Eclipse plugin `http://sourceforge.net/projects/eclint/`

DIY Lint

```
g++ -c \                                -Wcast-qual \
 -O2 \                                  -Wcast-align \
 -pedantic \                            -Wconversion \
 -ansi \                                -Waggregate-return \
 -Wall \                                -Wpacked \
 -Wformat=2 \                           -Wpadded \
 -Wmissing-include-dirs \               -Wlong-long \
 -Wswitch-enum \                        -Wvariadic-macros \
 -Wswitch-default \                     -D_POSIX_SOURCE=1 \
 -Wunused-parameter \                   -D_ISOC99_SOURCE=1 \
 -Wextra \                              -D_REENTRANT \
 -Wfloat-equal \                        -Dlint \
 -Wno-endif-labels \                   -D__NO_STRING_INLINES\
 -Wshadow \                             "$@" -o /dev/null
 -Wpointer-arith \
```

Further Tools

- static analysis tools
  - Coverity: `http://www.coverity.com`
  - Klocwork: `http://www.klocwork.com`
- instrumenting compilers (dynamic analysis)
  - Insure++: `http://www.parasoft.com`
- free memory debugging tools
  - valgrind: `http://valgrind.org`
  - Electric Fence: `http://directory.fsf.org/ElectricFence.htm`

Summary

- do not fix bugs — avoid them!
- focus on interesting stuff — let the computer handle the annoying details
- use the tools intelligently — they are often much more useful than you think
- pay attention to what your tools tell you — they are not there to just annoy you
- your tools are not as intelligent as you are — investigate the root cause of each problem flagged, usually it is a design issue
- think of the maintainer — it may be a good friend of yours or it may be you next time
- think in the future tense!