Advanced practical **Programming** for Scientists

Thorsten Koch

Zuse Institute Berlin

TU Berlin

WS2014/15

Difference between

```
#include <reader.h>  and #include “reader.h”    -I
```

Int Overflow: Baer, I.Gamrath, Lang, Steger, Weltsch

```
printf("\nError: invalid character '%c' in line %d\nSpecify number of rows and columns first", *p, line_num);
            exit(EXIT_FAILURE);  -> abort() -> assert(false) / assert(0);
```

```
./prog ../../../../ex5/data3/err_overflow.dat ff
prog: src/linear_program.c:59: add_constraint_lp: Assertion `lp->constr + 2 < lp->max_constr' failed.
Aborted (core dumped)
```

```
src/num_type.c:25:9: warning: format '%ld' expects argument of type 'long int', but argument 3 has type 'num_t' [-Wformat]
```

Compiler Flags

Standard flags

```
-c
-o output_file
–llibrary -Lpath
-Ipath –Ddefine=xxx -Udefine
-O
```

Special flags gcc

```
-std=c99 -Wall -Wextra –Wpedantic
-Wuninitialized
-Wbad-function-cast -Wcast-qual -Wcast-align -Wwrite-strings

-Wsuggest-attribute=pure -Wsuggest-attribute=const -Wsuggest-
attribute=noreturn
```

-Wstrict-overflow=n

This option is only active when -fstrict-overflow is active. It warns about cases where the compiler optimizes based on the assumption that signed overflow does not occur. Note that it does not warn about all cases where the code might overflow: it only warns about cases where the compiler implements some optimization. Thus this warning depends on the optimization level.

An optimization that assumes that signed overflow does not occur is perfectly safe if the values of the variables involved are such that overflow never does, in fact, occur. Therefore this warning can easily give a false positive: a warning about code that is not actually a problem. To help focus on important issues, several warning levels are defined. No warnings are issued for the use of undefined signed overflow when estimating how many iterations a loop requires, in particular when determining whether a loop will be executed at all.

-Wstrict-overflow=1

Warn about cases that are both questionable and easy to avoid. For example, with -fstrict-overflow, the compiler simplifies x + 1 > x to 1. This level of -Wstrict-overflow is enabled by -Wall; higher levels are not, and must be explicitly requested.

-Wstrict-overflow=2

Also warn about other cases where a comparison is simplified to a constant. For example: abs (x) >= 0. This can only be simplified when -fstrict-overflow is in effect, because abs (INT_MIN) overflows to INT_MIN, which is less than zero. -Wstrict-overflow (with no level) is the same as -Wstrict-overflow=2.

-Wstrict-overflow=3

Also warn about other cases where a comparison is simplified. For example: x + 1 > 1 is simplified to x > 0.

-Wstrict-overflow=4

Also warn about other simplifications not covered by the above cases. For example: (x * 10) / 5 is simplified to x * 2.

-Wstrict-overflow=5

Also warn about cases where the compiler reduces the magnitude of a constant involved in a comparison. For example: x + 2 > y is simplified to x + 1 >= y. This is reported only at the highest warning level because this simplification applies to many comparisons, so this warning level gives a very large number of false positives.

-Wfloat-equal

Warn if floating-point values are used in equality comparisons.

The idea behind this is that sometimes it is convenient (for the programmer) to consider floating-point values as approximations to infinitely precise real numbers. If you are doing this, then you need to compute (by analyzing the code, or in some other way) the maximum or likely maximum error that the computation introduces, and allow for it when performing comparisons (and when producing output, but that's a different problem). In particular, instead of testing for equality, you should check to see whether the two values have ranges that overlap; and this is done with the relational operators, so equality comparisons are probably mistaken.

-Wfloat-equal

```
bip_enum.c:173:24: warning: comparing floating point with == or != is unsafe [-
Wfloat-equal]
             assert(val == 0);
                        ^
bip_enum.c:245:46: warning: comparing floating point with == or != is unsafe [-
Wfloat-equal]
                   || (int_coef && rintl(val) != val)
                                              ^
In file included from bip_enum.c:10:0:
bip_enum.c: In function ‘bip_is_feasible’:
bip_enum.c:408:22: warning: comparing floating point with == or != is unsafe [-
Wfloat-equal]
          assert(x[c] == 0.0 || x[c] == 1.0);
                      ^
bip_enum.c:426:18: warning: comparing floating point with == or != is unsafe [-
Wfloat-equal]
          if (lhs == bip->b[r])
                  ^
```

-Wsign-conversion

```
splitline.c:52:30: warning: conversion to ‘size_t’ from ‘int’ may change the sign
of the result [-Wsign-conversion]
       lfs->field = calloc(lfs->size, sizeof(*lfs->field));
                              ^
```

```
#define LFS_INITIAL_SIZE   100
lfs->size  = LFS_INITIAL_SIZE;
lfs->field = calloc(lfs->size, sizeof(*lfs->field));
```

```
CFLAGS    = -std=c99 -Wall -Wextra -Wpedantic -Wshadow \
            -Wwrite-strings -Wuninitialized \
            -Wbad-function-cast -Wcast-qual -Wcast-align

CFLAGS    +=  -g

# gcov
CFLAGS    +=  -g -fprofile-arcs -ftest-coverage

# valgrind
CFLAGS    +=  -g -O

CFLAGS    +=  -g -O3 -march=native
```

char* t;

const char* s;

strtod(const char*, char**)

strtod(s, &t)

Where is t pointing to ?

Initializing/freeing const see max_coef

Other compilers

gcc

clang

icc

Exercise

Extend the program from the previous week in the following way:

- Make it faster
- Fix issues left from this week
- Add Makefile (if not already there)
- Make a coverage analysis with gcov
- Remove all warnings
- Find bugs in the example code.