From ISO/IEC 9899:1999 to 9899:201x

C is quirky, flawed, and an enormous success. – Dennis M. Ritchie

Florian Westphal fw@strlen.de

Hagen Paul Pfeifer hagen.pfeifer@protocollabs.de

February, 2011

Introduction

- WG14 Charter Principles (none are absolute):
  - Existing code is important, existing implementations are not
  - C code can be portable – C code can be non-portable
  - Avoid “quiet changes”
  - Keep the spirit of C
    - Trust the programmer
    - Don’t prevent the programmer from doing what needs to be done
    - Keep the language small and simple.
    - Provide only one way to do an operation
    - Make it fast, even if it is not guaranteed to be portable
    - Make support for safety and security demonstrable
  - Support international programming (additional principle for C9X)

- Minimize incompatibilities with C++ (additional principle for C9X)
- Trust the programmer is outdated (additional principle for C1X, programmers need the ability to check their work)

Introduction

- JTC1/SC22 is the international standardization subcommittee for programming languages
  - WG14 – C
  - WG9 – Ada
  - WG21 – C++
  - WG23 – Programming Language Vulnerabilities
  - ...

Introduction C99 - Determining the Type of a Literal Constant

- C99
  - Decimal constant (no suffix): `int, long int, long long int`
  - Decimal constant (l or L suffix): `long int, long long int`
- But: to be upward compatible with C89 and C++ the list should be:
  - `int, long int, unsigned long int, long long int`
  - `long int, unsigned long int, long long int`
- . . . but it isn't!
- Consequence (arch: `i386-pc-linux-gnu` (32 bit)):
  - In C99 4000000000 fits into `long long`
  - In C89 4000000000 fits into `unsigned long`
  - Result: C99 and C89 are not compatible
  - `(4000000000 > -1) ? "> - C99" : "< - C89"`

Introduction C99 - Comments in C89 / C99

- C99 added support for C++ -Style Comments ("//")
- . . . alters program behaviour:

```
printf("%d\n", 1 //* */ 2
);
```

Run Time Assertions

- `assert(3)` is run-time checked
- `#include <assert.h>; void assert(scalar expression);`
- Implemented as a macro:
  - Statement with side-effects are triggered with `NDEBUG`
  - Never use assert() with side-effect statements
- But: some expression can be checked at compile time - no need for run-time overhead!
- `if ((sizeof(struct foo) % 23) != 0) die("foo size error");`

Compile Time Assertions

- Linux Kernel:
  - `BUILD_BUG_ON(sizeof(struct foo) % 23));`
  - `#define BUILD_BUG_ON(condition) ((void)sizeof(char[1 - 2*!!(condition)]))`
    - `char[1]` or `char[-1]`
    - `sizeof()` is there to actually declare no array
    - The cast prevents the compiler from generating an warning message
- Another idea (Miguel Sofer):
  - `#define ct_assert(e) {enum { ct_assert_value = 1/(!!(e)) };}`
- C++:
  - `static_assert(constant-expression, "error message");`
  - `_Static_assert ( constant-expression , string-literal ) ;`

Path of no Return

- Especially for libraries: `longjump`, `raise`, `abort`, `fatal()`, `die()`, `abort()`, ...
- makes faster code, because the compiler can optimize more aggrisavly and produce less machine code
- static analysis tool can provide more usefull feedback (if code path after a `_Noreturn` is there, a `_Noreturn` declared function returns a value)

```
void f (void) {
  FILE *f;
  f = fopen( file, ...);
  if (f == NULL) {
    handle_error( ... );
  }
  /* work with f */
}

_Noreturn void f () {
  abort(); /* ok */
}
```

- handle_error() - a analysis tool cannot make sure if handle_error return or not. If declared as `_Noreturn` then it is a lot simpler. And not: handle_error may be linked

in, there is probably no source code available for the analysis tool.

- If a function is called where the function was previously declared with the noreturn attribute and the function returns anyway the behavior is undefined
- `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1453.htm`

Exclusive Access

- Till C99 no exclusive access `fopen()`
  - Open must fail if a file already exist

```
FILE *fp = fopen("foo.txt","r");
if( !fp ) {
  /* file does not exist */
  fp = fopen("foo.txt","w");
  ...
  fclose(fp);
} else {
  /* file exists */
  fclose(fp);
}
```

- Race Condition between both fopen calls!
- Open Group Base Specification: `O_CREAT` and `O_EXCL`
- `open("foo.txt", O_CREAT | O_EXCL)` fails if file already exists
- C1X: add X flag: `wx` (create text file for writing with exclusive access)

Transparent Struct and Union

```
struct packet {
        struct {
                int proto; /* AF_INET or AF_INET6 */
                union {
                        unsigned int addr_v4;
                        unsigned char  addr_v6[16];
                };
        } header;
        char *data;
        size_t data_len;
}
struct packet pkt = { .header { .proto = AF_INET, .addr_v4 = INADDR_ANY },
.data = NULL, data_len = 0 };
```

C1x specifcation outline

- some of the new functionality is optional → ”conditional feature macros”, e.g.:

  `__STDC_ANALYZABLE__`: annex L conformance

  `__STDC_IEC_559__`: Floating point arithmetic handling in annex F

- Thread support is also optional (`__STDC_NO_THREADS__`)

Generic Selection

- New keyword: `_Generic`
- Selects assignment expressions based on type names

```
#define cbrt(X) _Generic((X)), long: cbrtl, default cbrt ) (X)
```

expands to `cbrtl(X)` if `X` is of type `long` and `cbrt(X)` otherwise.

Thread support

- `#include <threads.h>`
- Nobody would use C's threads if they were wildly different to POSIX threads, which are already widely used, documented and understood
- Similar to pthreads: mutexes, condition variables, ...
- `pthread_mutex_init` → `mtx_init`, ...
- Several pthread features unsupported, e.g. `pthread_setschedprio`
- Unlike pthreads, c1x contains functions for atomic operations and memory ordering
- `_Thread_local` storage-class specifier: `_Thread_local int myvar;`

Atomic Ops

- `#include <stdatomic.h>`
- New type qualifier: `_Atomic`: `_Atomic int foo;` `stdatomic.h`
- Defines Types, Macros and Functions:
  - Atomic Types: `atomic_char, atomic_short, atomic_int, ...`
  - Operations on atomic types (init, compare, add, sub, . . .)
  - `enum memory_order`: synchronize memory accesses
  - „*Fences*” to order loads/stores

Atomicity

- `i++` is not an *atomic* operation
- Value has to be fetched, modified, and written back to memory
- For single-threaded applications this is not relevant (exception: signal handler)
- . . . but when concurrent access to `i` is possible, this is no longer true
- c1x makes it possible to perform such modifications in a single operation

Atomic Functions

- `atomic_{load,store,exchange}`: assignments, swap values, ...
- `atomic_fetch_{add,sub,or,xor,and}`: modify atomic_ type, returns new value
- `atomic_compare_exchange_`: atomic conditional swap, i.e. `if (a == b) a = c;` The result of the comparison is returned.
- fences: synchronization operation without a memory location

Ordering

```
void foo(void) {
 int a, b;
 [..]
 a = 42;
 b = 23;
 [..]
```

The compiler or the CPU is free to re-order the assignments; there are no side-effects.

Ordering

- 2nd example. Lets consider adding an element to a linked list.
  1. The element is initialized (setting `->next` to `NULL`, etc)
  2. The element is assigned: `tail->next = elem`
- Q: Could the compiler re-order this?
- Q: Could the CPU re-order this?

Memory Ordering

- Several of the atomic functions are so-call "synchronization operations".
- Necessary to make changes to memory locations in one thread visible to others in a reliable fashion.
- Several stdatomic.h functions also have a corresponding `_explicit` version, e.g. `atomic_load(_Atomic *a)`→`atomic_load_explicit(_Atomic *a, memory_order m)`
- version without `_explictit` has `memory_order_seq_cst` semantics
- `memory_order`: enum that defines memory ordering constraints:
  - `memory_order_relaxed`: no ordering
  - `memory_order_acquire,release`: load/store (read/write)
  - `memory_order_seq_cst`: "Sequential Consistency": single total order for all accesses to all variables.

Fences (barriers)

```
void atomic_thread_fence(memory_order m)
void atomic_signal_fence(memory_order m)
```

- Compiler Optimizations & load/store reordering are inhibited
- `thread_fence` also emits HW fence instructions, `signal_fence` does not
- depending on `memory_order`, affects loads, stores or both

Alignment

- gcc:
  - `__attribute__((aligned( )))`
- `posix_memalign()`
  - since glibc 2.1.91
- C1X: `#include <stdalign.h>`
  - New keywords: `_Alignas`, `alignof`
  - New macros:
    - `alignof`: returns the alignment requirements of the operand type
    - `alignas`: is used to force stricter alignment requirements
  - New function: `aligned_alloc()`

Bounds Checking Interfaces

- C1X Draft Annex K
- libc run-time constraint checks
- `set_constraint_handler_s()`
- `abort_handler_s, ignore_handler_s`
- New functions with `_s` suffix:

  `fopen_s, fprintf_s, strcpy_s, strcat_s, gets_s, ...`

Unicode Support

- In the past
  - All characters were the same size
  - 8 bit (7 bit)
  - string.h provides utilities function (english)
- → UNICODE
- C1X adds new datatypes
  - `char16_t` (UTF-16)
  - `char32_t` (UTF-32)
  - (C++0x compatible)
- String Literals
  - `u""`
  - `U""`

Complex Numbers

- C99 – "Complex types were added to C as part of the effort to make C suitable and attractive for general numerical programming." (Rationale for International Standard – Programming Languages C)
  - `float _Complex`
  - `double _Complex`
  - `long double _Complex`
- Macros to create/modifiy complex numbers
- `#include <complex.h>`
- `int i = 3.5;`
- `double complex c = 5 + 3 * I;` (Macro I expands to `_Imaginary_I` or `_Complex_I`)
- Conditional feature:
  - `__STDC_NO_COMPLEX__` implementation does not support complex types

- Alignment requirement as an array containing exactly two elements of corresponding real type (`{float, double, long double} [2]`)
- Operation of double `_Complex` and `float` yield to double `_Complex`
- Trigonometric Functions
  - `double complex cacos(double complex z);`
  - `long double complex cacosl(long double complex z);`
  - `long double complex cpowl(long double complex x, long double complex y);`
  - `double creal(double complex z);`
- Section: `7.3 Complex arithmetic`

Quick Exit

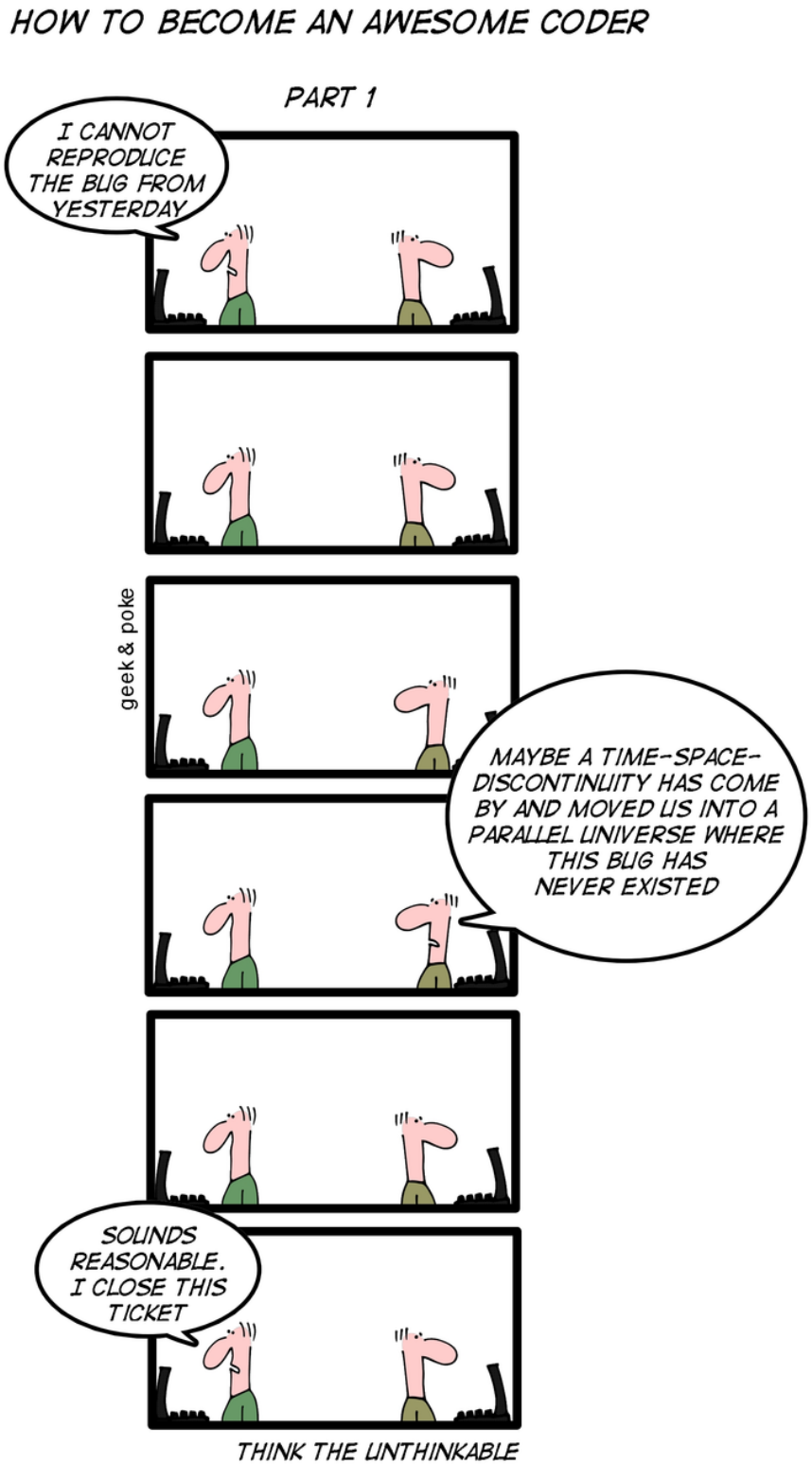
- 7.22.4.7
- Synopsis:

```
#include <stdlib.h>

_Noreturn void quick_exit(int status);
```

- causes normal program termination to occur
- functions registered by the atexit function are not called
- signal handlers registered by the signal function are not called
- longjmp() results in undefined behavior
- glibc 2.10 (2009-03-08 Ulrich Drepper):
  - `stdlib/quick_exit.c`
  - `stdlib/at_quick_exit.c`

Questions?


HOW TO BECOME AN AWESOME CODER
PART 1
I CANNOT REPRODUCE THE BUG FROM YESTERDAY
geek & poke
MAYBE A TIME-SPACE-DISCONTINUITY HAS COME BY AND MOVED US INTO A PARALLEL UNIVERSE WHERE THIS BUG HAS NEVER EXISTED
SOUNDS REASONABLE. I CLOSE THIS TICKET
THINK THE UNTHINKABLE

Overflows

- `abs(x) >= 0` is not always true!
- `abs(INT_MIN) = -2147483648`
- If `fstrict-overflow` is enabled an expression like `abs(x) >= 0` can be simplified to a constant expression - be aware!
  - The compiler will assume that when doing arithmetic with signed numbers overflow will not happen
  - Since GCC 4.2 this option is on by default with `-O2`, `-O3` and `-Os`
- `Wstrict-overflow=2` to warn about simplifications