The C Language
An International Standard

CIS2750

Professional Aspect of Software Engineering

ANSI C

- The C language appeared in the early 1970s and was originally cryptic, small, and tolerant. For the next ten years C evolved in the UNIX environment.
- The resultant language was first described in the paper “Portability of C Programs and the UNIX System,” by S.C. Johnson and Dennis Ritchie in 1978.
- This style of C was commonly called UNIX C.

ANSI C

- In 1989 the American National Standards Institute (ANSI) standardized C as standard X3.159-1989.
  - In December of that year ISO adopted the ANSI C standard making minor changes.
- In 1990 ANSI then re-adopted ISO standard C.
  - This version of C is known as either **ANSI C** or Standard C or **C89**
- ISO/IEC 9899:1999 = “C99”, influenced by C++
  - Adopted by ANSI in 2000
- ISO/IEC 9899:2011 = “C11” → newest standard

Implementation Characteristics

- The ANSI standard defines several terms which describe the characteristics of an implementation. They are useful to describe **what is and is not acceptable** in the language.
- **Implementation defined** code means that the compiler writer chooses what happens and has to document it.
  - *Example:* whether the sign bit is propagated when shifting a bit right.

Implementation Characteristics

- **Unspecified behavior** for something correct which the standard does not impose any requirements.
  - *Example:* the order of argument evaluation.
- **Undefined behavior** is something incorrect which the standard does not impose any requirements. Anything may happen, from nothing, to a warning message, to program termination.
  - *Example:* what happens when a signed integer overflows.

Implementation Characteristics

- A **constraint** is a restriction or requirement which must be obeyed. If you do not then your program will become undefined.
  - *Example:* the operands of the % operator must be of integral type or a diagnostic will result.
- An interesting problem with the definition of constraints is that compilers only have to produce an **error** message if a program violates both syntax and constraints.

Implementation Characteristics

- **Semantic rules** which are not explicitly stated can be broken and because this behavior is undefined the compiler does not have to issue a **warning**.
  - *Example:* the C standard header files have a function called **malloc**, but redefining this function is not a constraint so the compiler does not have to warn if this happens.

Implementation Characteristics

- **Strictly conforming** code is one which:
  - uses only specified features of the language
  - doesn't exceed any implementation-defined limits
  - has no output that depends on implementation defined, unspecified, or undefined features
- Highest chance of success for porting between compilers/systems without changes!

Implementation Characteristics

Conforming programs *can* depend on **non-portable** features.

- A conforming program is considered with respect to a **specific** implementation and may be non-conforming using a different compiler.

- Compilers *can* have extensions, but not ones which alter the behavior of a strictly conforming program.

ANSI Changes to (older) C

- A variety of minimum sizes were defined by the standard including:
  - 31 parameters in a function definition
  - 31 arguments in a function call
  - 509 characters in a source line
  - 32 levels of nested parentheses in an expression
  - **long** integers are at least 32 bits

New Features of C99

- Coding
  - end-of-line (//) comments like C++
  - Mix declarations and code
  - Remove implicit function declaration
- Data types
  - **long long int** and **long double** (minimum 64 bits)
  - **_Complex** → pair of floating point, <complex.h> ops
  - **_Bool** boolean aka **bool**, <stdbool.h> with **true** & **false**
  - Variable length arrays
  - Flexible array members of structures
- Library
  - sprintf family of functions in stdio.h

New Features of C11

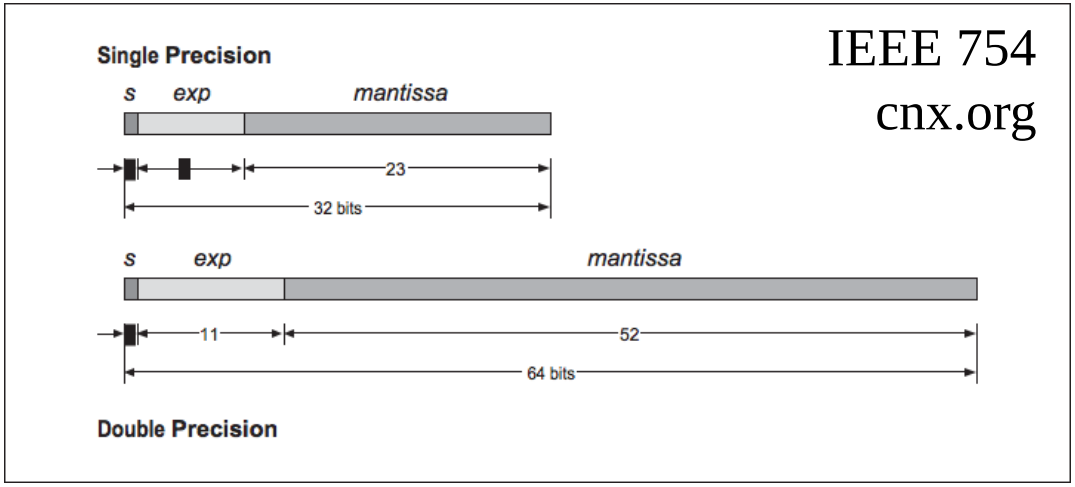
- Latest 2011 standard, available on cis2750.socs
  - Was called C1X; goes along with C++11
  - Some feature are “optional” for compilers to support
- Data types
  - Unicode chars: **char16_t** & **char32_t** for UTF-16/32
- “Security” **_s** versions of exploitable risky functions
  - gets() → gets_s(), others like this
- Multithreading built in <threads.h> <stdatomic.h>
- Array bounds checking (optional feature)

Lengths of Numeric Data Types

- Lengths (in bits) *not* defined by C language
  - Cp. Java: byte, short (2 bytes), int (4), long (8)
- **C integers:** sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)
  - **char** must have at least 8 bits, **short** and **int** 16+, **long** 32+, **long long** 64+
  - More bits ➔ greater negative/positive range
  - NOTE: char considered a *numeric* type
    - We conveniently use it for 8-bit character codes like ASCII and UTF-8

Floating Point Data Types

- **C floating point:** sizeof(float) <= sizeof(double) <= sizeof (long double)
  - Aka “single” vs. “double” vs. “quad precision”
- Scientific notation stored in 3 bit-fields:
  - s = sign bit (0 pos., 1 neg.);
  - exp = (signed) exponent → mantissa * 2^{exp}



IEEE 754
cnx.org

Floating Point Data Types

- Single vs. double precision, more bits ➔
  - greater negative/positive range of exponent (can represent greater magnitudes) ~ 10^{38} vs. 10^{308}
  - greater precision of mantissa (can represent more significant digits)
- Tricks to save bits
  - *Normalization:* shift mantissa left/right (adjusting exp) till it starts with "1." then don't store "1" (assume it)
  - Add "bias" of 127 (or 1023) to exp to avoid having to store negative exponents

Floating Point Arithmetic

- Multiply and divide are easy:
  - $x*2^a * y*2^b = x*y*2^{a+b}$
  - $x*2^a / y*2^b = x/y*2^{a-b}$
- Add and subtract require aligning binary points
  - shift mantissas left/right, adjusting exps till equal
  - then add/subtract mantissas
  - NOTE: when magnitudes of numbers differ greatly *or* subtracting numbers that are very close, “round-off error” can cause loss of significance
- Pitfalls of f.p. arithmetic forms its own big topic

Why Do We Care about Older C?

- For new development, we don't, but…
- Huge installed code base is "old"
  - Many jobs involve **maintenance** of legacy code
  - Avoid misunderstanding legacy code
    - Based on your new-C habits
  - Avoid introducing bugs into legacy code
    - Keep to legacy coding style for consistency!
- Choices: retain older C compiler *or* try to upgrade to new compiler standard (exposes "code rot")

What is “code rot”?

- Aka “software rot” “bit rot”
- When program hasn’t been changed, yet stops working/compiling
  - Earlier assumptions no longer valid and program breaks
    - Famous Y2K bug → 2 bytes no longer enough, storage-saving shortcut induced bogus operation
  - Compiler, libraries, or OS “upgraded,” features discontinued or incompatible

Holding Code Rot at Bay

- Recognize assumptions that won't stand test of time
  - "Users will *never* need more than 32767 records, so 16-bit index is fine!" (+1 looks <0) → *Comair fiasco*
- Write strictly-conforming software, not relying on extensions or "tricks"
  - E.g., intentionally accessing outside array bounds
- Recompile when language spec revised, system upgraded, etc.
  - May discover small problems, easily fixed
  - Or BIG problems needing major reexamination