Computer Programming

Internal representation. Bitwise operators

Marius Minea

marius@cs.upt.ro

21 October 2014

Ideal math and C are not the same!

In mathematics:
- integers $\mathbb{Z}$ and reals $\mathbb{R}$ have *unbounded* values (are infinite)
- reals are *dense* (have *infinite precision*)

In C:
- numbers take up *finite* memory space (a few bytes)
- $\Rightarrow$ have *finite range*; reals have *finite precision*

To correctly work with numbers, we must understand:
- their representation and storage in memory
- their size and precision limitations
- what *overflow* and *rounding* errors may appear

Memory representation of objects

Any value (parameter, variable, also constant) needs to be represented in memory and takes up some program space

bit = unit of data storage that may hold two values (0 or 1)
need not be individually addressable (usually is not)

byte = addressable unit of data storage that may hold a character
in C: `CHAR_BIT` $\geq$ 8 bits (`limits.h`)
8 bits in all usual architectures

`sizeof` *operator*: size of a type or value in *bytes*
`sizeof(`*type*`)` or `sizeof` *expression*

`sizeof(char)` is 1: *a character takes up one byte*
also unicode and wide character support: `uchar.h`, `wctype.h`
an integer has `sizeof(int)` *bytes* $\Rightarrow$ `CHAR_BIT`*sizeof(int) *bits*

`sizeof` is an *operator*, NOT function; evaluated at compile-time

Binary representation of numbers

In memory, numbers are represented in binary (base 2)

unsigned integers, with N bits

$c_{N-1}c_{N-2}\ldots c_1c_0\ {}_{(2)} = c_{N-1}\cdot 2^{N-1} + \ldots + c_1\cdot 2^1 + c_0\cdot 2^0$

c_{N-1} = *most significant* (higher-order) bit

c_0 = *least significant* (lower-order) bit

Range of values: from 0 to $2^N - 1$ e.g. 11111111 is 255

$c_0 = 0 \Rightarrow$ *even* number; $c_0 = 1 \Rightarrow$ *odd* number

signed integers: allowed representations: i) sign-magnitude

ii) *two's complement*: sign bit is -2^{N-1} *used in practice*

iii) one's complement: sign bit is $-(2^{N-1} - 1)$

$\Rightarrow$ Range for two's complement is from -2^{N-1} to $2^{N-1} - 1$

$0c_{k-2}\ldots c_1c_0\ {}_{(2)} = c_{k-2}\cdot 2^{k-2} + \ldots + c_1\cdot 2^1 + c_0\cdot 2^0$ (≥ 0)

$1c_{k-2}\ldots c_1c_0\ {}_{(2)} = -2^{k-1} + c_{k-2}\cdot 2^{k-2} + \ldots + c_0\cdot 2^0$ (< 0)

Examples (8 bits):

11111111 is -1 11111110 is -2 10000000 is -128

Integer types

Before the type `int` one can write *specifiers* for:

size: `short`, `long`, since C99 also `long long`

sign: `signed` (implicit, if not present), `unsigned`

Can be combined; may omit `int`: e.g. `unsigned short`

`char`: `signed char` [-128, 127] or `unsigned char` [0, 255]
`int, short`: $\geq$ 2 bytes, at least [-2^{15} (-32768), $2^{15}-1$]
`long`: $\geq$ 4 bytes, at least [-2^{31} (-2147483648) , $2^{31}-1$]
`long long`: $\geq$ 8 bytes, at least [-2^{63}, $2^{63}-1$]

Corresponding *signed* and *unsigned types* have the same size:
`sizeof(short)`$\leq$`sizeof(int)`$\leq$`sizeof(long)`$\leq$`sizeof(long long)`

`limits.h` defines names (macros) for limits, e.g.
`INT_MIN`, `INT_MAX`, `UINT_MAX`, likewise for `CHAR`, `SHRT`, `LONG`

since C99: `stdint.h`: fixed-width integers in two's complement
`int8_t`, `int16_t`, `int32_t`, `int64_t`,
`uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`

Use sizeof to write portable programs!

Sizes of types are *implementation dependent*
(processor, OS, compiler ...)
⇒ use `sizeof` to find storage taken up by a type/variable

DON'T write programs assuming a given type has 2, 4, 8, ... bytes
program will *run incorrectly* on other systems

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

int main(void)
{ // below, z is printf format modifier for sizeof
  printf("Integers have %zu bytes\n", sizeof(int));
  printf("Smallest (negative) int: %d\n", INT_MIN);
  printf("Largers (positive) unsigned: %u\n", UINT_MAX);
  return 0;
}
```

Integer constants

base 10: as usual, e.g., `-5`
base 8: prefixed by 0 (zero): `0177` (127 decimal)
base 16: prefixed by `0x` or `0X`: e.g., `0x1aE` (430 decimal)
Can't write in any other base. *Can't write binary* ~~1101110~~.
suffixes: `u` or `U` for `unsigned`, e.g., `65535u`
`l` or `L` for `long` e.g., `0177777L`,   `ll` or `LL` for `long long`

Character constants

printable characters between single quotes: `'0'`, `'!'`, `'a'`

special characters:		`'\0'`	nul	`'\a'`	alarm
`'\b'`	backspace	`'\t'`	tab	`'\n'`	newline
`'\v'`	vert. tab	`'\f'`	form feed	`'\r'`	carriage return
`'\"'`	double quote	`'\''`	quote	`'\\'`	backspace

characters written in octal (max. 3 digits), e.g., '\14'
characters written in hexadecimal (prefix `x`), e.g., `'\xff'`

The `char` type is *an integer type* (of smaller size)
Char constants are *automatically converted* to `int` in expressions.
(this is why you don't see functions with `char` parameters)

What use are bitwise operators ?

To access the internal representation of data (e.g., numbers) and represent/encode/process some types of data efficiently

A *set* (of integers): can use a bit for each possible element
(1 = is member; 0 = is not member of set)
⇒ sets of small integers: using an int (`uint32_t`, `uint64_t`)
(fixed-width integer types defined in `stdint.h`)

Set operations:
intersection = bitwise AND
union = bitwise OR
adding an element: setting the corresponding bit

The *current date* can be represented using bits:
day: 1-31 (5 bits); month: 1-12 (4 bits)
year: 7 bits suffice for 1900 to 2027
⇒ need operations to extract day/month/year from a 16-bit value (e.g. `uint16_t`)

Bitwise operators

Can *only* be used for *integer* operands!

`&`	bitwise AND	(1 only if both bits are 1)
`|`	bitwise OR	(1 if at least one of the bits is 1)
`^`	bitwise XOR	(1 if *exactly* one of the bits is 1)
`~`	bitwise complement	(opposite value: $0 \leftrightarrow 1$)

`<<` left shift with number of bits in second operand
vacated bits are filled with zeros; leftmost bits are lost

`>>` right shift with number of bits in second operand
vacated bits filled with zero if number is unsigned or nonnegative
else implementation-dependent (usually repeats sign bit)
$\Rightarrow$ *for portable code, only right-shift* `unsigned`

All operators work with *all bits* independently
They *don't change operands*, just give a result (like `+`, `*`, etc.)

Examples

```
     01101010        01101010        01101010
&    10101101   |    10101001   ^    10101101
-------------   -------------   -------------
     00101000        11101011        11000111

~    01101010   11101010u  >> 2   11101010  << 2
-------------   ---------------   --------------
     10010101   00111010u         10101000
```

only right-shift `unsigned`

Checking individual bits

Use a *mask* (integer value) with only one bit 1 in desired position

1) shift mask, keep number in place

```
void printbits1(unsigned n) { // ~(~0u>>1) = 1000...0000
  for (unsigned m = ~(~0u>>1); m; m >>= 1)
    putchar(n & m ? '1' : '0');
}
```

2) constant mask, shift number

```
void printbits2(unsigned n) {
  for (int m = 1; m; m <<= 1, n <<= 1) // m counts bit width
    putchar(n & ~(~0u>>1) ? '1' : '0');
}
```

3) same, but directly check sign bit

```
void printbits3(unsigned n) {
  for (int m = 1; m; m <<= 1, n <<= 1)
    putchar((int)n < 0 ? '1' : '0');
}
```

Properties of bitwise operators

`n << k` has value $n \cdot 2^k$ (if no overflow)
`n >> k` has value $n/2^k$ (integer division) for unsigned/nonnegative
`1 << k` has 1 only in bit `k` $\Rightarrow$ is 2^k for `k < 8*sizeof(int)`
$\Rightarrow$ use this, *not* `pow` (which is floating-point!)
`~(1 << k)` has 0 only in bit k, rest are 1

`0` has all bits 0, `~0` has all bits 1 (= -1, since it's a signed int)
`~` preserves signedness, so `~0u` is unsigned (`UINT_MAX`)

`&` with 1 preserves a bit, `&` with 0 is always 0
`n & (1 << k)` *tests* (is nonzero) bit k in `n`
`n & ~(1 << k)` *resets* (makes 0) bit k in the result

`|` with 0 preserves a bit, `|` with 1 is always 1
`n | (1 << k)` *sets* (to 1) bit k in the result

`^` with 0 preserves value, `^` with 1 flips value
`n ^ (1 << k)` *flips* bit k in result

Again, *none of these have side effects*, they just produce results.

Creating and working with bit patterns (masks)

& with 1 preserves   & with 0 resets
| with 0 preserves | with 1 sets

Value given by bits 0-3 of `n`: AND with $0\ldots01111_{(2)}$ `n & 0xF`
Reset bits 2, 3, 4: AND with `~`$0\ldots011100_{(2)}$ `n &= ~0x1C`
Set bits 1-4: OR with $11110_{(2)}$ `n |= 0x1E` `n |= 036`
Flip bits 0-2 of `n`: XOR with $0\ldots0111_{(2)}$ `n ^= 7`
⇒ choose fitting operator and *mask* (easier written in hex/octal)

Integer with all bits 1: `~0` (signed) or `~0u` (unsigned)
k rightmost bits 0, rest 1: `~0 << k`
k rightmost bits 1, rest 0: `~(~0 << k)`
`~(~0 << k) << p` has k bits of 1, starting at bit p, rest 0
`(n >> p) & ~(~0 << k)`: `n` shifted `p` bits, reset all except last `k`
`n & (~(~0 << k) << p)`: reset all except `k` bits starting at bit `p`

More about identifiers: linkage and `static`

We have discussed *scope* (visibility) and *lifetime* (storage duration)
Linkage: how do same names in different scopes/files link ?

Identifiers declared with `static` keyword have *internal linkage*
(are not linked to objects with same name in other files)
Storage duration if declared `static` is lifetime of program.

`static` in function: local scope but preserves value between calls!
initialization done only once, at start of lifetime

```
#include <stdio.h>
int counter(void) {
  static int cnt = 0;
  return cnt++;
}
int main(void) {
  printf("counter is %d\n", counter()); // 0
  printf("counter is %d\n", counter()); // 1
  return 0;
}
```

Representing real numbers

Similar to *scientific representation* known from school in base 10:
$6.022 \cdot 10^{23}$, $1.6 \cdot 10^{-19}$: leading digit, decimals, exponent of 10

In computer: *base 2*; *sign, exponent and mantissa* (significand)

$$(-1)^{sign} * 2^{exp} * 1.mantissa_{(2)}$$

Bit pattern: `S EEEEEEEE MMMMMMMMMMMMMMMMMMMMMMM`
IEEE 754 floating point format (used by most implementations):
`float`: 4 bytes: 1+8+23 bits; `double`: 8 bytes: 1+11+52 bits
exponent represented *in excess of* a *bias*/offset (127 for float):
for $0 < \mathrm{E} < 255$ we have $(-1)^S * 2^{E-127} * 1.M_{(2)}$
for $\mathrm{E} = 0$, small (denormalized) numbers: $(-1)^S * 2^{-127} * 0.M_{(2)}$
also: representations for ± 0, $\pm\infty$, errors (NaN)

C standard also specifies rounding directions, exceptions/traps, etc.

Floating point precision

Precision of real numbers is *relative* to their absolute value
(*floating* point rather than *fixed* point)

e.g. smallest `float` > 1 is $1 + 2^{-23}$ (last bit of mantissa is 1)
For larger numbers, *absolute* imprecision grows
e.g., $2^{24} + 1 = 2^{24} * (1 + 2^{-24})$, last bit does not fit in mantissa
$\Rightarrow$ will be rounded: not all integers can be represented as `float`

```
FLT_EPSILON 1.19209290e-07F      // min. with 1+eps > 1
DBL_EPSILON 2.2204460492503131e-16  // min. with 1+eps > 1
```

Real types

C imposes $sign \cdot (1 + mantissa) \cdot 2^{exp}$ format and some size / precision limits (need not be IEEE 754)
$\Rightarrow$ value range is symmetric w.r.t. zero

Sample *limits* from `float.h`:
`float`: 4 bytes, ca. 10^{-38} to 10^{38}, 6 significant digits
`FLT_MIN 1.17549435e-38F FLT_MAX 3.40282347e+38F`
`double`: 8 bytes, ca. 10^{-308} to 10^{308}, 15 significant digits
`DBL_MIN 2.2250738585072014e-308 DBL_MAX 1.7976931348623157e+308`
`long double`: for higher precision (12 bytes)

Floating-point constants: with decimal point, optional sign and exponent (prefix `e` or `E`); integer or fractional part may be missing:
`2.` `.5` 1.e-6, .5E+6
Implicit type: `double`; sufix `f`, `F`: `float`; `l`, `L`: `long double`

Use `double` for sufficient precision in computations!
`math.h` functions: `double`; variants with suffix: `sin`, `sinf`, `sinl`

Watch out for overflows and imprecision!

`int` (even `long`) may have small range (32 bits: $\pm$ 2 billion)
Not enough for computations with large integers (factorial, etc.)
Use `double` (bigger range) or arbitrary precision libraries (bignum)

Floating point has limited precision: beyond 1E16, `double` does not distinguish two consecutive integers!

A decimal value may not be precisely represented in base 2:
may be periodic fraction: $1.2_{(10)} = 1.(0011)_{(2)}$
`printf("%f", 32.1f);` writes 32.099998

Due to precision loss in computation, result may be inexact
$\Rightarrow$ replace `x==y` test with `fabs(x - y) <` *small epsilon*
(depending on the problem)

Differences smaller than precision limit cannot be represented:
$\Rightarrow$ for `x < DBL_EPSILON` (ca. 10^{-16}) we have `1 + x == 1`

Usual arithmetic conversions (implicit)

In general, the rules go from larger to smaller types:

1. if an operand is `long double`, convert the other to `long double`
2. if any operand is `double`, the other is converted to `double`
3. if any operand is `float`, the other is converted to `float`
4. perform *integer promotions*: convert `short`, `char`, `bool` to `int`
5. if both operands have signed type or both have unsigned type
   convert smaller type to larger type
6. if unsigned type is larger, convert signed operand to it
7. if signed type can fit all values of unsigned type, convert to it
8. otherwise, convert to unsigned type corresponding to operand
with signed type

(negative) `int` becomes `unsigned` in operation with `unsigned`

```
unsigned u = 5;
if (-3 > u) puts("what?!"); // -3u == UINT_MAX - 2
```

Explicit and implicit conversions

Implicit conversions (summary of previous rules)
integer to floating point, smaller type to larger type
integer promotions: `short`, `char`, `bool` to `int`
when equal size, convert to unsigned

Conversions in assignment: truncated if lvalue not large enough

```
char c; int i; c = i;  // loses higher-order bits of i
```

!!! Right-hand side evaluated *independently* of left-hand side!!!

```
unsigned eur_rol = 43000, usd_rol = 31000 // currency
double eur_usd = eur_rol / usd_rol; // result is 1 !!!
```

(integer division happens before assignment to double)

Floating point is truncated towards zero when assigned to int
(fractional part disappears)

Explicit conversion (type cast): (*typename*) *expression*
converts expression as if assigned to a value of the given type

```
eur_usd = (double)eur_rol / usd_rol  // int to double
```

Watch out for sign and overflows!

WARNING `char` may be `signed` or `unsigned` (implementation dependent, check `CHAR_MIN`, is either 0 or `SCHAR_MIN`)
⇒ different values in conversion to `int` if bit 7 is 1
`getchar/putchar` work with `unsigned char` converted to `int`

WARNING: most any arithmetic operation can cause overflow

```
printf("%d\n", 1222000333 + 1222000333); // -1850966630
```

(if 32-bit, result has higher-order bit 1, and is considered negative)

```
printf("%u\n", 2154000111u + 2154000111u); // truncated: 4032926
```

CAREFUL when comparing / converting `signed` and `unsigned`

```
if (-5 > 4333222111u) printf("-5 > 4333222111 !!!\n");
```

because −5 converted to `unsigned` has higher value

Correct comparison between `int i` and `unsigned u`:
`if (i < 0 || i < u)` or `if (i >= 0 && i >= u)`
(compares `i` and `u` only if `i` is nonnegative)
Check for overflow on integer sum `int z = x + y`:

```
if (x > 0 && y > 0 && z < 0 || x < 0 && y < 0 && z >= 0)
```

ERRORS with bitwise operators

DON'T right-shift a negative int!

~~`int n = ...; for ( ; n; n >>= 1 ) ...`~~

May loop forever if `n` negative; the topmost bit inserted is usually the sign bit (implementation-defined). Use `unsigned` (inserts a 0).

DON'T shift with more than bit width (behavior undefined)