Computer Programming

Internal representation. Bitwise operators

Marius Minea

marius@cs.upt.ro

23 October 2017

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)
- ⇒ have *finite range*; reals have *finite precision*

To correctly work with numbers, we must understand:

- representation and storage in memory
- *size* limitations ⇒ *overflow* errors
- *precision* limitations ⇒ *rounding* errors

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 (can't refer to just one bit)

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

the `sizeof` *operator*: gives size of a type or value in *bytes* not ~~bits~~
`sizeof(`*type*`)` or `sizeof` *expression*

`sizeof(char)` is 1: *a character takes up one byte*
for unicode and wide character support: `uchar.h`, `wctype.h`
an `int` has `sizeof(int)` *bytes* ⇒ `CHAR_BIT*sizeof(int)` *bits*
All ints, big (10000) and small (5) use `sizeof(int)` bytes!

`sizeof` is NOT a function; evaluated (if possible) at compile-time

Binary representation of ints: two's complement

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

unsigned integers: for N bits, value is computed as

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

c_{N-1} = *most significant* (higher-order) bit (MSB)
c_0 = *least significant* (lower-order) bit (LSB)
Range of values: from 0 to 2^N-1 e.g. 11111111 is 255
LSB $c_0 = 0 \Rightarrow$ *even* number; $c_0 = 1 \Rightarrow$ *odd* number

signed integers: MSB is sign; N-1 bits value: several encodings
i) sign-magnitude: if MSB is 1, take value part as negative
ii) one's complement: sign bit counts as $-(2^{N-1}-1)$
iii) *two's complement (used in practice)*: sign bit counts as -2^{N-1}
$\Rightarrow$ Range for two's complement is from -2^{N-1} to $2^{N-1}-1$
$1c_{N-2}\dots c_1c_0\ _{(2)} = -2^{N-1} + c_{N-2}\cdot 2^{N-2} + \dots + c_0\cdot 2^0 \qquad (< 0)$

unsigned: 0..255 $\Rightarrow$ *signed*: 0..127 + 128..255 become $-128..-1$
8-bit: 11111111 is -1 11111110 is -2 10000000 is -128

Integer types: choose the right size

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, must cover $[-2^{15}$ (-32768), $2^{15}-1]$
`long`: $\geq$ 4 bytes, must cover $[-2^{31}$ (-2147483648) , $2^{31}-1]$
`long long`: $\geq$ 8 bytes, must cover $[-2^{63}, 2^{63}-1]$

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

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

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

Integer and char constants: base 8, 10, 16

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: w/ 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	`’\\’`	backslash

octal (max. 3 digits): '\14' *Caution* type **`char`** may be **`signed`**

hexadecimal (prefix `x`): '\xff' `0xFF`: int 255, '\xff' may be -1

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 ?

access the *internal representation* of data (e.g., numbers)

efficiently encode information (e.g. header fields in network packets or files; status values/commands from/to hardware)

efficient data structures: sets of small integers
one bit per element (1 = is member; 0 = is not member of set)
one 32-bit int for any set of ints 0..31 (4 billion combinations)

	intersection	bitwise AND
Set operations:	union	bitwise OR
	add element	set corresponding bit

date/time can be represented using bits:
min/sec (0-59): 6 bits   hour (0-23): 5 bits   day (1-31): 5 bits
month (1-12): 4 bits   year: 6 bits left from 32: 1970-2033
⇒ need operations to get day/month/year from 32-bit value

Bitwise operators

Can *only* be used for *integer* operands! Not ~~`float`~~!

All operators work with *all bits* independently (not just one bit!)

`&` 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` numbers

Examples

$$\begin{array}{cr} & 01101010 \\ \& & 10101101 \\ \hline & 00101000 \end{array} \qquad \begin{array}{cr} & 01101010 \\ | & 10101001 \\ \hline & 11101011 \end{array} \qquad \begin{array}{cr} & 01101010 \\ \text{\textasciicircum} & 10101101 \\ \hline & 11000111 \end{array}$$

$$\begin{array}{cr} \text{\textasciitilde} & 01101010 \\ \hline & 10010101 \end{array} \qquad \begin{array}{ll} 111010\cancel{11}\text{u} & >>\ 2 \\ \hline 00111010\text{u} & \end{array} \qquad \begin{array}{ll} \cancel{11}101010 & <<\ 2 \\ \hline 10101000 & \end{array}$$

only right-shift **`unsigned`** numbers!

Bit operators *don't change operands*, they just give a result

If `x` is 7, `x+2` is 9, but `x` is still 7. Only `x = x+2` changes `x` !

Bitwise operators are no different!

`x & 0xF` or `x >> 2` will compute some results, `x` will be the same!

Printing a number in octal (base 8)

```
void printoct(unsigned n)
{
  if (n > 8) printoct(n/8);
  putchar('0' + n % 8);
}
```

$8 = 2^3 \Rightarrow$ Each octal digit corresponds to a group of 3 bits.

e.g. one hundred is 0...001 100 100 $(8^2 + 4 \cdot 8 + 4)$

$\Rightarrow$ can use bit operators to isolate parts

```
void printoctbits(unsigned n)
{
  unsigned n1 = n >> 3; // ``shift out'' last digit
  if (n1) printoct(n1); // not all bits are zero
  putchar('0' + (n & 7)); // & 7 (111) gives last 3 bits
}
```

Likewise, can use groups of 4 bits to obtain hex digits

careful to get either `'0'..'9'` or `'A'..'F'` for printing

Working with individual bits

Bitwise operators work with *all* bits.

But, if choosing the appropriate operation and operand (“mask”) we can *check* / *set* / *reset* / *flip* a single bit

`1 << k` : bit `k` is 1, rest 0

`&` with 1 gives same bit, `&` with 0 is always 0

`n & (1 << k)` *tests* bit k of `n` (is nonzero?)

`n & ~(1 << k)` *resets* (makes 0) bit k in the result

`|` with 0 gives same 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

Printing 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

`1 << k` : bit `k` is 1, rest 0 ⇒ is 2^k for `k < 8*sizeof(int)`

`n << k` has value $n \cdot 2^k$ (if no overflow)

`n >> k` has value $n/2^k$ (integer division) for unsigned/nonnegative
⇒ use this, *not* `pow` (which is floating-point!)

`~(1 << k)` only bit k is 0, 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`)

Bit ops produce results (like +, *, etc), *without changing operands*

Only *assignment* operators (and pointer dereference) change values!

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 $\tilde{}0\ldots011100_{(2)}$ `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;
}
```

Working with real numbers

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` suffix `f`, `F`: **`float`**; `l`, `L`: **`long double`**

Implicit type of floating constants: **`double`**.

`float` function arguments are promoted to **`double`**

e.g. in calls to `printf`, where `"%f"` means **`double`**

Real types have limited range and precision!

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 range and precision (12 bytes)

Representing real numbers

Similar to *scientific/normalized notation* in base 10:
$6.022 \cdot 10^{23}$, $1.6 \cdot 10^{-19}$: *leading digit* ($\neq 0$), decimals, exp. of 10

In computer: *base 2*; *sign, exponent and mantissa* (significand)
$(-1)^{sign} * 2^{exp} * 1.mantissa_{(2)}$ $1 \leq 1.mantissa < 2$

IEEE 754 floating point format (used by most implementations)
Bit pattern: `S EEEEEEEE MMMMMMMMMMMMMMMMMMMMMMM`

`float`: 4 bytes: 1+8+23 bits; **`double`**: 8 bytes: 1+11+52 bits

exponent represented *in excess of* a *bias* (float: 127, double: 1023)

number is $\boxed{(-1)^S \cdot 2^{E-127} \cdot 1.M_{(2)}}$ for $0 < \mathrm{E} < 255$

Caution! the 1 before mantissa is *implicit* (not in bit pattern)

E=0: small numbers, $\pm 2^{-126} \cdot 0.M_{(2)}$ E=255: ±`INFINITY`, `NAN`

Working with bit representation of float

$9.75 = 9\frac{3}{4} \quad \lfloor \log_2 9.75 \rfloor = 3 \Rightarrow 9.75 = 2^3 \cdot 1\frac{7}{32} \quad \frac{7}{32} = .00111_{(2)}$

`float` is $\underbrace{0}_{\text{sign}}\ \underbrace{10000010}_{127+3}\ \underbrace{0011100\ldots0}_{\text{23-bit mantissa}}$

Extracting the mantissa M as `unsigned` (low-order 23 bits)
and adding the implicit 1 on bit 23: `M1 = 1 << 23 | M`
then the number is $(-1)^S \cdot 2^{E-127} \cdot \mathtt{M1} \cdot 2^{-23}$
i.e., the mantissa part is $(2^{23} + \mathrm{M}) \cdot 2^{-23} = 1 + \mathrm{M} \cdot 2^{-23}$

Floating point has limited 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 1 bit does not fit in mantissa
$\Rightarrow$ **`float`** can represent 2^{24} and $2^{24} + 2$, but $2^{24} + 1$ is rounded up

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

E $= 0$: 0 and small (denormal) numbers: $(-1)^S * 2^{-126} * 0.M_{(2)}$
E $= 255$: ±`INFINITY`, `NAN` (not-a-number, error)

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

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

Watch out for overflows and imprecision!

`int` (even `long`) may have small range (32 bits: ± 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
⇒ replace test `x==y` with `fabs(x - y) <` *small epsilon*
(depending on the problem)

Differences smaller than precision limit cannot be represented:
⇒ 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
```

compile with `-Wconversion` and `-Wsign-compare` or `-Wextra`

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`: 0 or `SCHAR_MIN`)
⇒ different `int` conversion if bit 7 is 1 (`'\xff'` = `-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); //overflow: 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)

AND with a one-bit mask is not 0 or 1, but 0 or nonzero

`n & (1 << k)` is either 0 or `1 << k`