COMP10002

Semester One, 2018

Number Representations

Week 10, lecturer: Jianzhong Qi

Slides developed by Alistair Moffat

Overview

Algorithmic goals

Number representations

Unsigned types

Other radixes

Floating point

The preprocessor

Chapter 13 – Concepts

All data types end up being represented as bits stored in bytes and words.

Each numeric type has a different representation.

Each numeric type has advantages and disadvantages.

All computer arithmetic is limited in some way or another.

Algorithmic objectives

For symbolic processing (for example, sorting strings), desire algorithms that are:

- Above all else, correct
- Straightforward to implement
- Efficient in terms of memory and time
- (For massive data) Scalable and/or parallelizable
- (For simulations) Statistical confidence in answers and in the assumptions made.

Algorithmic objectives

For numeric processing, desire algorithms that are:

- Above all else, correct
- Straightforward to implement
- Effective, in that yield correct answers and have broad applicability and/or limited restrictions on use
- Efficient in terms of memory and time
- (For approximations) Stable and reliable in terms of the underlying arithmetic being performed.

The last one can be critically important.

Numeric problems

Wish to compute

$$f(x) = x \cdot \left(\sqrt{x+1} - \sqrt{x}\right)$$

and

$$g(x) = \frac{x}{\sqrt{x+1} + \sqrt{x}} \,.$$

- sqdiff.c

Hmmmm, why did that happen?

not overflow
number representation

for large x
$\sqrt{x+1} - \sqrt{x}$ dif is tiny
since few place to represent double 16 bytes
float 8 bytes

Numeric problems

Wish to compute

$$h(n) = \sum_{i=1}^{n} \frac{1}{i}$$

- `logsum.c`

Hmmmm, why did that happen?

Arithmetic – Pitfalls

start from large to ↓ small

doesn't allow precision

start from small ↓ to large

prepare for precision

In all numeric computations need to watch out for:

- subtracting numbers that are (or may be) close together, because absolute errors are additive, and relative errors are magnified
- adding large sets of small numbers to large numbers one by one, because precision is likely to be lost
- comparing values which are the result of floating point arithmetic, zero may not be zero.

And even when these dangers are avoided, numerical analysis may be required to demonstrate the convergence and/or stability of any algorithmic method.

Numbers – Bits, bytes, and words

Inside the computer, everything is stored as a sequence of binary digits, or bits.

Each bit can take one of two values – "0", or "1".

A byte is a unit of eight bits, and most computers a word is a unit of either four or eight bytes.

A word typically stores a set of 32 or 64 bits. The interpretation of that bit sequence depends on the type of the variable involved, and the representation used for the different data types.

Binary numbers

In `char`, `short`, `int`, and `long` variables, the bits are used to create a binary number.

In decimal, the number 345 describes the calculation $3 \times 10^2 + 4 \times 10^1 + 5 \times 10^0$.

Similarly, in binary, the number 1101 describes the computation $1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$, or thirteen in decimal.

Binary numbers

Binary counting: 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111, 10000, and so on.

With a little bit of practice, you can count to 1,023 on your fingers; and with a big bit of practice, to 1,048,575 if you use your toes as well. (Be careful with 4 and 6.)

There are two further issues to be considered:

- negative numbers, and
- the fixed number of bits w in each word.

Integer representations

In an unsigned $w = 4$ bit system, the biggest value than can be stored is 1111, or 15 in decimal.

Adding one then causes an integer overflow, and the result 0000.

The second column of Table 13.3 (page 232) shows the complete set of values associated with a $w = 4$ bit unsigned binary representation.

When $w = 32$, the largest value is $2^{32} - 1 = 4{,}294{,}967{,}295$.

Integer representations

if want to represent
negative number,
set the first bit
to be the
sign

Bit pattern	Integer representation		
	unsigned	sign-magn.	twos-comp.
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	−0	−8
1001	9	−1	−7
1010	10	−2	−6
1011	11	−3	−5
1100	12	−4	−4
1101	13	−5	−3
1110	14	−6	−2
1111	15	−7	−1

Sign-magnitude representation

To handle negative numbers, one bit could be reserved for a sign, and $w - 1$ bits used for the magnitude of the number.

The third column of Table 13.3 shows this sign-magnitude interpretation of the 16 possible $w = 4$-bit combinations.

There are two representations of the number zero.

Adding one to `INT_MAX` gives -0.

Twos-complement representation

The final column of Table 13.3 shows twos-complement representation. In it, the leading bit has a weight of $-(2^{w-1})$, rather than 2^{w-1}.

If that bit is on, and $w = 4$, then subtract $2^3 = 8$ from the unsigned value of the final three bits.

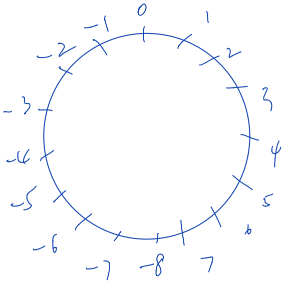
So 1101 is expanded as
$1 \times -(2^3) + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$, which is minus three.

Twos-complement representation

The advantages of twos-complement representation are that

- there is only one representation for zero, and
- integer arithmetic is easy to perform.

For example, the difference $4 - 7$, or $4 + (-7)$, is worked out as $0100 + 1001 = 1101$, which is the correct answer of minus three.



Twos-complement representation

$2^4 \quad 2^3 \quad 2^1$

$26 = 16 + 8 + \dots 2 + 0$

$0\ 0\ 0\ 1\ 1\ 0\ 1\ 0$

$-26 \quad 1\ 1\ 1\ 0\ 0\ 1\ 1\ 0$

从右往左 保留最后一个1，以及之后的0，

其他的 $1 \to 0$

$0 \to 1$

On a $w = 32$-bit computer the range is from $-(2^{31}) = -2{,}147{,}483{,}648$ to $2^{31} - 1 = 2{,}147{,}483{,}647$. Beyond these extremes, `int` arithmetic wraps around and gives erroneous results.

If $w = 64$-bit arithmetic is used (type `long long`), the range is $-(2^{63})$ to $2^{63} - 1 = 9{,}223{,}372{,}036{,}854{,}775{,}807$, approximately plus and minus nine billion billion, or 9×10^{18}

The type `char` is also an integer type, and using 8 bits can store values from $-(2^7) = -128$ to $2^7 - 1 = 127$

	-128	64	32	16	8	4	2	1
-26	1	1	1	0	0	1	1	0

Unsigned types

C offers a set of alternative integer representations, `unsigned char`, `unsigned short`, `unsigned int` (or just `unsigned`), `unsigned long`, and `unsigned long long`.

Negative numbers cannot be stored.

But will get printed out if you still use `"%d"` format descriptors. Use `"%u"` instead, or `"%lu"`, or `"%llu"`.

Unsigned types

C also provides low-level operations for isolating and setting individual bits in `int` and `unsigned` variables.

These operations include left-shift (`<<`), right-shift (`>>`), bitwise and (`&`), bitwise or (`|`), bitwise exclusive or (`^`), and complement (`~`).

There are some subtle differences between `int` and `unsigned` when bit shifting operations are carried out.

Unsigned types

Figure 13.1 on page 233 shows some of these operations in action.

- `intbits.c`

Table 13.5 (page 236) gives a final precedence table that includes all of the bit operations. If in doubt, over parenthesize.

Octal and hexadecimal

C also supports constants that are declared as octal (base 8) and hexadecimal (base 16) values. Beware! Any integer constant that starts with 0 is taken to be octal:

```
int n1 = 020;
int n2 = 0x20;
printf("n1 = %oo, %dd, %xx\n", n1, n1, n1);
printf("n2 = %oo, %dd, %xx\n", n2, n2, n2);
```

gives

```
n1 = 20o, 16d, 10x
n2 = 40o, 32d, 20x
```

Octal and hexadecimal

The standard Unix tool `bc` can be used to do radix conversions:

```
mac: bc
ibase=10
obase=2
25
11001
obase=8
25
31
obase=16
25
19
```

Floating point representations

The floating point types `float` and `double` are stored as:

- a one bit sign, then
- a w_e-bit integer exponent of 2 or 16, then
- a w_m-bit mantissa, normalized so that the leading binary or hexadecimal digit is non-zero.

Floating point representations

When $w = 32$, a `float` variable has around $w_m = 24$ bits of precision in the mantissa part. This corresponds to about 7 or 8 digits of decimal precision.

In a `double`, around $w_m = 48$ bits of precision are maintained in the mantissa part.

Floating point representations

For example, when $w = 16$, $w_s = 1$, $w_e = 3$, $w_m = 12$, the exponent is a binary numbers stored using w_e-bit twos-complement representation, and the mantissa is a w_m-bit binary fraction:

Number (decimal)	Number (binary)	Exponent (decimal)	Mantissa (binary)	Representation (bits)
0.5	0.1	0	.100000000000	0 000 1000 0000 0000
0.375	0.011	−1	.110000000000	0 111 1100 0000 0000
3.1415	11.001001000011···	2	.110010010000	0 010 1100 1001 0000
−0.1	−0.0001100110011 ···	−3	.110011001100	1 101 1100 1100 1100

The exact decimal equivalent of the last value is −0.0999755859375. Not even 0.1 can be represented exactly using fixed-precision binary fractional numbers.

Floating point representations

Floating point representations can be investigated by casting a pointer.

- `floatbits.c`

$2^7 + 2^6 + 2^5 + \cdots 2^1$

$\frac{1-2^7}{1-2}$

Reveals that:

sign exponent mantissa

0.0	is	00000000 00000000 00000000 00000000
1.0	is	00111111 10000000 00000000 00000000
–1.0	is	10111111 10000000 00000000 00000000
2.0	is	01000000 00000000 00000000 00000000
10.5	is	01000001 00101000 00000000 00000000
20.1	is	01000001 10100000 11001100 11001101

More types

The (non-ANSI) extended types `long long` (64-bit integer) and `long double` (128-bit floating point value) might also come in useful at some stage.

The preprocessor

Lines that commence with a `#` are regarded as directives to the preprocessor.

Facilities provided include:

- symbolic substitution via `#define`
- parameterized "string replacement" substitution in `#define` definitions and expansions
- conditional compilation via `#if` and `#ifdef`
- access to compile-time variables.
- plus lots more.

Chapter 13 – Program examples

Use the force wisely, Luke!

- `preproc.c`

Chapter 13 – Summary

Everything is stored as bits.

If you understand how, you will be a better programmer.

The preprocessor is another powerful tool in the C toolkit.