Programming and Algorithms

Representation of numbers in machines.

Jean-Bernard Hayet

CIMAT, A.C.

Data in memory

- Computer memory (RAM): series of **bytes** (octets, in general).
- These bytes can be **addressed** through a unique address in memory.
- This address is specified by **a number of 64 bits** (in the case of 64-bit operating systems).
- When you run a program, you have access to a part of this memory.

Data types

$$\text{byte} \neq \text{bit}$$

addressable unit of data storage large enough to hold any member of the basic character set of the execution environment.

- comes from "to bite". The 'y' is used to avoid confusion with bit,
- in general, 1 byte = 1 octet = 8 bits.

Data types

To check how a byte is defined in you system, refer to the file limits.h:

```
> more /usr/include/limits.h
#ifndef _LIBC_LIMITS_H_
#define _LIBC_LIMITS_H_ 1

#define __GLIBC_INTERNAL_STARTING_HEADER_IMPLEMENTATION
#include <bits/libc-header-start.h>

/* These assume 8-bit ‘char’s, 16-bit ‘short int’s,
and 32-bit ‘int’s and ‘long int’s.  */

/* Number of bits in a ‘char’. */
#  define CHAR_BIT 8
```

Data types

- Machines process several bytes at the same time (this depends on its physical registers).
- These packets of bytes are called **words**.
- Most recent machines handle **64-bit** words (8 bytes).
- In general, k-bit systems have registers and buses that manipulate k bits (in hardware), and OS that manipulate k-bit addresses (software).
- One can have **64-bit machines with a 32-bit OS** (or 32-bit programs) but the opposite is not possible.

Data types

In C/C++, the **type** allows to specify a storage class for data and defines two things:

- the number of octets that will be used in memory for storing the data;
- the way the data are encoded into this chunk of memory.

The basic types are characters, integers and floating numbers (or floats, for real numbers).

There is no standard in terms of the size for each type.

The sizeof function

In C/C++, this is the function that gives the number of bytes necessary of a type/variable that s passed as arguments:

sizeof (int),
sizeof (x)

where x is the name of variable.

Data types

This is what is guaranteed about sizes of fundamental types:

```
1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long
sizeof(float) <= sizeof(double) <= sizeof(long double)
sizeof(I) == sizeof(signed I) == sizeof(unsigned I)
```

where I can be char, short, int or long. In addition, it is guaranteed that a char has at least 8 bits, a short at least 16 bits and a long at least 32 bits... assuming more is hazardous.

[The C++ Programming Language, B. Stroustrup, 3rd Edition]

Data types

In a typical 64-bit machine (mine):

```
sizeof char 1
sizeof short 2
sizeof int 4
sizeof long 8
sizeof long long 8
```

Or:
https://en.cppreference.com/w/cpp/types/integer.

Integer types

To represent numbers in a subset of $\mathbb{N}$ ("unsigned"): in base 2, **over n bits**

$$a = \sum_{i=0}^{n-1} a_i 2^i.$$

Range?
Convenient to use Hexadecimal.

Integer types

Examples:

- **unsigned char**, over 8 bits:
- **unsigned short**, over 16 bits:
- **unsigned int**, over 32 bits:

Integer types

Examples:

- **unsigned char**, over 8 bits: [0,255]
- **unsigned short**, over 16 bits:
- **unsigned int**, over 32 bits:

Integer types

Examples:

- **unsigned char**, over 8 bits: [0,255]
- **unsigned short**, over 16 bits: [0,65535]
- **unsigned int**, over 32 bits:

Integer types

Examples:

- **unsigned char**, over 8 bits: [0,255]
- **unsigned short**, over 16 bits: [0,65535]
- **unsigned int**, over 32 bits: [0,4294967295]

Integer types: Relative integers

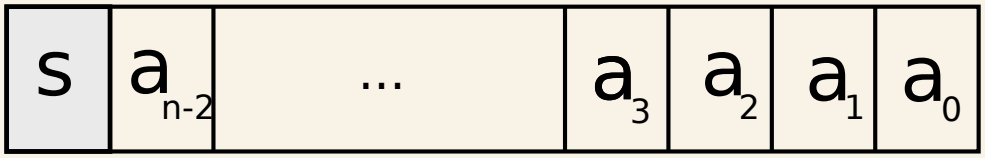
How to represent numbers in a subset of $\mathbb{Z}$ ("signed")?

We would like to:

- Take a negative from a positive in a fast way.
- For $a > 0$, $-(-a) = a$, and $a + (-a) = 0$.
- Use subtractions as additions (bitwise with carry-over):

$$a + (-b) = a - b.$$

Integer types: Relative integers



If we handle an n-bit integer type (8, 16, 32...), one option could be to assign:

- The sign s through one bit (the most significant).
- $s = 0$ for positive numbers (fast checking).

$$a = (1 - 2s) \sum_{i=0}^{n-2} a_i 2^i.$$

Integer types: Relative integers

Limits?

- taking the opposite is easy...
- what do we get by summing a number and its opposite?
- we cannot directly use the addition operator for summing numbers with opposite signs.

Integer types: Relative integers

One's complement, for $0 \leq a < 2^{n-1}$: we keep the structure above, but

$$a = |a| \triangleq \sum_{i=0}^{n-1} a_i 2^i \text{ with } a_{n-1} = 0$$

$$-a \triangleq \sum_{i=0}^{n-1} (1 - a_i) 2^i = 2^n - 1 - |a|.$$

- Opposite: easy.
- Testing sign: easy.
- Problems?

Integer types: Relative integers

Two's complement, for $a > 0$

$$-a \triangleq \sum_{i=0}^{n-1}(1 - a_i)2^i + 1 = 2^n - |a|.$$

- Taking the opposite: easy (same operation).
- Checking sign is easy.
- No distinction between substraction and addition.
- Only one zero to handle! (why?)
- Range?
- How to represent 131 and -131 as short?

Integer types: Relative integers

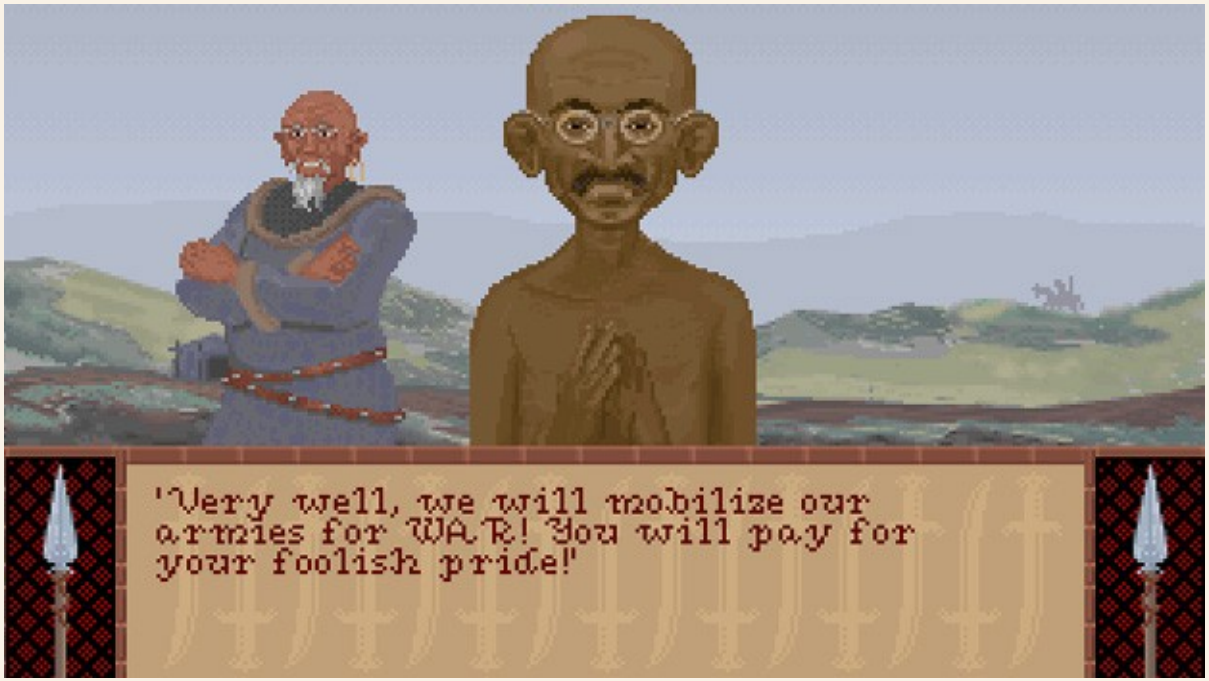
Two's complement:

- After the first one (which is left unchanged with the first zeros): take the complement to one of all the digits.
- In a processor, typically arithmetic operations can be monitored because the carry-over and the overflow status are stored.

Integer types: Overflow

- Situation when the result of an arithmetic operation **cannot ne represented in the type currently in use**.
- Example: with unsigned char, 230+57 gives an overflow.
- With the addition, overflow detection: how to do it easily?
  - Unsigned integers?
  - Signed integers?

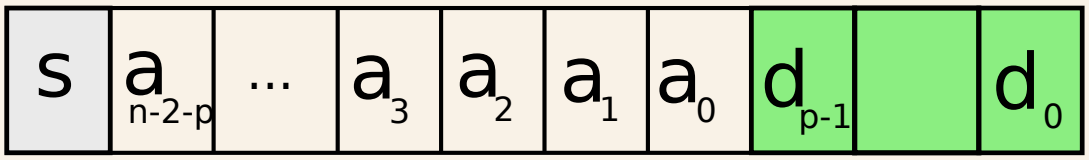
Integer types: Typical bug




```
unsigned char gandhi_aggressiveness = 1;
 ...
if (gandhi_adopts_democracy)
        gandhi_aggressiveness-=2;
```

Effect?

Fixed point representation



- Fixed point: Is used when one wants to work with real numbers at some given absolute precision.
- Representation :

$$z = (1 - 2s).a + d2^{-p}$$

- Range?
- There is no fixed-point support by default in C (need to use specialized libraries).

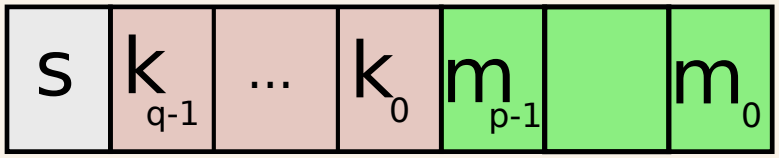
Fixed point representation

Example: Represent 78.3125 (decimal) with $n = 16$ and $p = 6$.

Fixed point representation

+ They allow to work at an arbitrary precision.
+ Fast to manipulate (similar to integers).
- Not very flexible.
- One wastes many bits for small numbers or numbers with few numbers after the point.
- For reaching a resolution of $\frac{1}{65536} \approx 1.510^{-5}$, you need 16 bits after the point.

Floating point types



- Defined in the IEEE 754 standard.
- Representation as signed/mantissa/exponent in a given base:

$$f = (1 - 2s).m.b^e \quad (1)$$

$$e = k - (2^{q-1} - 1) \quad (2)$$

$$0 \leq m < b \quad (3)$$

Ambiguity?

Floating point types

The standard defines two types:

Type	Sign	Exponent	Mantissa
float	1	8	23
double	1	11	52

Floating point types

Normalized mantissa: only one non-zero digit before the point (=1 if b = 2)

1.0101001
8.8178582

In binary, we have effectively $p = 24/53$ bits to represent the mantissa with $1 \leq m < 2$ and $b = 2$, as the leading digit 1 is **implicit**.

$$f = (1 - 2s).(1 + m.2^{-p}).2^{e}$$

Range of the exponent:

$$-2^{q-1} + 1 \leq e \leq 2^{q-1}$$

Floating point types

Example : represent all the positive floating numbers with $b = 2$, $q = 2$, $n = 5$.

Floating point types

- Represent 0?
- The IEEE 754 norm defines three additional numbers/objects: NaN, $+\infty$, $-\infty$ to handle operations like 1/0. . .
- How to represent them, in addition of the 0 ?

Solution: use the extreme values of the exponent.

Floating point types

Case of float (single precision floating number):

$e = k - 127$	m	f
$-126 \leq e \leq 127$	$0 \leq m < 2^{23}$	$\pm 1.m \times 2^e$
128	0	$\pm\infty$
128	$\neq 0$	NaN
-127	0	± 0
-127	$\neq 0$	$\pm 0.m \times 2^{-126}$

Floating point types

Case of double (double precision floating number):

$e = k - 1023$	m	f
$-1022 \leq e \leq 1023$	$0 \leq m < 2^{52}$	$\pm 1.m \times 2^e$
1024	0	$\pm\infty$
1024	$\neq 0$	NaN
-1023	0	± 0
-1023	$\neq 0$	$\pm 0.m \times 2^{-1022}$

Floating point types

- Observe the binary value that corresponds to 0.
- The cases $\pm\infty$ and NaN correspond to exponents 11111....
- With zero exponent (in binary) and non-zero mantissa, these are numbers in underflow (**not normalized**).

Floating point types: Limits

- Overflows.
- Underflows.
- Rounding errors.

Floating point types: Limits

With double, the smallest representable normalized floating number is

$$x = 1.0 \times 2^{-1022}.$$

Consider $y = (1 + 2^{-52}) \times 2^{-1022}$, can it be represented? Now, consider $x - y$, how is it represented? Then:

```
if (x!=y)
        z = 1.0/(x-y);
```

Test on $x - y$!

```
if (fabs(x-y)>epsilon)
        z = 1 . 0 / ( x-y ) ;
```

Floating point types: Limits

```
#include <stdio.h>
#include <math.h>
int main() {
        float f = pow(2.0,-23) ;
        float a = (1.0 )*pow(2,-125);
        float b = (1.0+f)*pow(2,-125), z;
        if (a !=b)
                fprintf ( stderr ,"a and b have different representations \n") ;
        else
                fprintf ( stderr ,"a and b have same representations \n") ;
        fprintf ( stderr ,"Values : %e %e %e \n" , a , b , f ) ;
        float d = a-b;
        z = 1.0f/d ;
        unsigned int id = *(unsigned int*)&d;
        fprintf ( stderr , "Diff : %e \n", d) ;
        fprintf ( stderr , "Diff : %x \n", id) ;
        unsigned int iz = *(unsigned int*)&z;
        fprintf ( stderr ,"Diff inv : %e \n", z);
        fprintf ( stderr ,"Diff inv : %x \n", iz );
        return 0;
}
```

Floating point types: Limits

```
a and b have different representations
Values : 2.350989e-38 2.350989e-38 1.192093e-07
Diff : -2.802597e-45
Diff : 80000002
Diff inv : -inf
Diff inv : ff800000
```

Interpretation?

Floating point types: Overflows

The first test of Ariane 5 in 1996 concluded with a crash.
The reason was:

- A velocity was stored as a double (64 bits), and then converted into a short in a small bit of code.
- Tests/simulations were done with dimensions of Ariane 4.
- During the flight, overflow on the short caused an exception.
- The exception was not processed, which caused a reset of the inertial sensors.
- Without these data, the rocket was programmed to crash.

More information here.

Floating point types: Rounding

Not all the reals can be represented correctly with floating point types: for example the decimal 0.1 needs an infinity of digits in a binary system. This implies a **rounding**!

```
int main() {
        for (double x = 0 ; x != 0.3 ; x += 0.1);
}
```

Floating point types: Rounding

Several ways for defining the rounding of a real x such that $x^- \leq x \leq x^+$, with x^- and x^- floating numbers in a given representation:

- Towards $-\infty$.
- Towards $+\infty$.
- Towards 0.
- Towards the closest (and in case of equality, take the float with even mantissa).

Floating point types: Correct rounding

- For a given function, **correct rounding** means that it gives you the same result as when one computes the result at infinite precision and then applies the rounding on the result.
- The norm **imposes correct rounding for addition, subtraction, multiplication, division and square root**.

Floating point types: Correct rounding

- With correct rounding, we can get to interesting mathematical analysis of successions of arithmetic operations: **We can express intervals in which the true result should belong to**.
- Portability.
- Operators such as log or cos are not included in the norm.

Floating point types: Rounding errors

In 1991, a Patriot missile battery was activated to intercept a Scud missile, during the First Gulf War. In the navigation system, the date/time was expressed as multiples of tenths of seconds.
To handle it as a date, an approximation of a tenth of second was used over 24 bits: 209715.2^{-21} , error of 10^{-7} seconds each tenth of seconds.
As the battery had been switched on 100 hours before, it had accumulated a $0.34s$ error and missed its target.
Balance : 28 dead.

Floating point types: Double Rounding

Consider $x = 1.0110100000001$.

- Rounding of x to the closest, 9 bits: $y = 1.01101000$
- Rounding of y to the closest, 5 bits: $z = 1.0110$
- Rounding of x to the closest, 5 bits : $z' = 1.0111$!!

This is the double rounding problem; can be a problem with some FPUs that work internally in extended double precision (64bits mantissa) and then convert in double.
http://www.vinc17.org/research/extended.en.html

Floating point types: Rounding error

The **absolute rounding error** is the error that corresponds to the last digit of the mantissa.

If r is the number of digits after the point, then, in the case of rounding to the closest :

$$(b/2) \times b^{-r-1} \times b^{e}.$$

Floating point types: ulp

This represents the same rounding error but evaluated at the last digit (unit in the last place). For example, if z is

$$d.dd...dd \times b^e,$$

the error in ulp is

$$|d.dd...dd - (z/b^e)|b^r$$

If $r = 3, b = 10$, what is the error in ulps to approximate .0314159 by 3.140×10^{-2}? by 3.142×10^{-2} ?
By definition, rounding to the closest floating point number has a maximum error of 0.5 **ulp**.

Floating point types: relative error

(Valor exacto-Valor representado)/(Valor exacto)

- What is the relative error that corresponds to rounding to the closest (.5 ulp)s?
- The values are between b^e y $b \times b^e$, hence the relative error is:

$$(1/2)b^{-r-1} \leq ulp \leq (b/2)b^{-r-1}$$

- A relative rounding error is always inferior to $\frac{1}{2}b^{-r}$ (**machine precision**).

Floating point types: Arithmetic operations

Additions/subtractions are done as follows:

1. Align both numbers along the number of higher exponent (bit shift).
2. Apply the operation on the mantissa.
3. Eventually, re-align to get a normalized result.

Floating point types: Cancellation

Example in base 10, $10.1 - 9.93$

$$x = 1.01 \times 10^1 \tag{4}$$
$$y = 0.99 \times 10^1 \tag{5}$$
$$x - y = 0.02 \times 10^1 \tag{6}$$

Error: 30 ulps! (and every digit is incorrect!)

Floating point types: Cancellation

This is a typical problem in computation: **cancellation**.

Differences between close numbers can eliminate the leading bits (which are the most confiable), leaving and "promoting" those which are less confiable (they may come with rounding errors).

Amplifying effect!

Floating point types: Cancellation

In the worst case, the relative error can be:

$$(b^{-r} - b^{-r-1})/b^{-r-1} = b - 1$$

Hence the absolute error **can be as high as the result!**

One trick to avoid the problem is to do the operation with one additional digit (guard digit). In this situation, one can show that the relative error cannot be larger than 2ϵ.

Floating point types: Cancellation

Consider the root formulas for 2nd grade polynomials

$$\frac{1}{2a}(-b+\sqrt{b^2-4ac}) \tag{7}$$

$$\frac{1}{2a}(-b-\sqrt{b^2-4ac}). \tag{8}$$

If a, b, c are rounded and $b^2 >> ac$, we have problems
More robust formulas :

$$\frac{2c}{-b-\sqrt{b^2-4ac}}, \frac{2c}{-b-\sqrt{b^2+4ac}}$$

Floating point types: Cancellation

What is the best : compute $x^2 - y^2$ or $(x - y)(x + y)$? Why?

Maybe useful to analyze the method/algorithm to evaluate its sensibility to cancellation problems.

Do not forget that the order of the operations is important (no associativity, no distributivity).

References

- V. Lefèvre y P. Zimmermann, Arithmétique flottante, Rapport de recherches INRIA N.5105.
- The GNU C Library, online manual.
- David Goldberg. What every computer scientist should know about floating-point arithmetic. ACM Computing Surveys, 23(1):5-48, March 1991.