using namespace cln;

CLN: A Class Library for Numbers

Richard B. Kreckel

Castro Urdiales, September 1 2006

3.1415926535897932384626433832795028841971693993751058209749445923078164062
862089986280348253421170679821480865132823066470938446095505822317253594081
284811174502841027019385211055596446229489549303819644288109756659334461284
756482337867831652712019091456485669234603486104543266482133936072602491412
737245870066063155881748815209209628292540917153643678925903600113305305488
204665213841469519415116094330572703657595919530921861173819326117931051185
480744623799627495673518857527248912279381830119491298336733624406566430860
213949463952247371907021798609437027705392171762931767523846748184676694051
320005681271452635608277857713427577896091736371787214684409012249534301465
495853710507922796892589235420199561121290219608640344181598136297747713099
605187072113499999983729780499510597317328160963185950244594553469083026425
223082533446850352619311881710100031378387528865875332083814206171776691473
035982534904287554687311595628638823537875937519577818577805321712268066130
019278766111959092164201989380952572010654858632788659361533818279682303019
520353018529689957736225994138912497217752834791315155748572424541506959508
295331168617278558890750983817546374649393192550604009277016711390098488240
128583616035637076601047101819429555961989467678374494482553797747268471040
475346462080466842590694912933136770289891521047521620569660240580381501935
112533824300355876402474964732639141992726042699227967823547816360093417216
412199245863150302861829745557067498385054945885869269956909272107975093029
553211653449872027559602364806654991198818347977535663698074265425278625518
184175746728909777727938000816470600161452491921732172147723501414419735685
481613611573525521334757418494684385233239073941433345477624168625189835694
855620992192221842725502542568876717904946016534668049886272327917860857843

Overview

- History of the library
- Feature Overview
- The Type System

  bringing mathematical types and OO together
- Selected Implementation Aspects
  - Implementing `class cl_I : public cl_RA ...`
  - Fixnums and Bignums
- Small Example

  printing the largest known perfect number
- Finally. . .

  Applications

CLN History

late 1980s-1995: arbitrary precision types within CLisp (Bruno Haible et al.)
`http://clisp.cons.org/` (1987–today)
implementation languages: C and Assembler

1995: spin-off from CLisp
implementation languages: C++ and Assembler
purpose: make the arbitrary precision numbers of CLisp available to a broader public

1996: option to base low-level routines on more efficient GMP routines
`http://www.swox.com/gmp/` (MPN level only)
computation of 1 000 000 decimal digits of $\zeta(3)$

1999-01-12: release of CLN version 1.0

2000: maintainer change, since Bruno Haible was busy doing CLisp, Linux I18N, Unicode Support (libiconv), GLibC and several other free software projects

CLN History

late 1980s-1995: arbitrary precision types within CLisp (Bruno Haible et al.)
`http://clisp.cons.org/` (1987–today)
implementation languages: C and Assembler

1995: spin-off from CLisp
implementation languages: C++ and Assembler
purpose: make the arbitrary precision numbers of CLisp available to a broader public

1996: option to base low-level routines on more efficient GMP routines
`http://www.swox.com/gmp/` (MPN level only)
computation of 1 000 000 decimal digits of $\zeta(3)$

1999-01-12: release of CLN version 1.0

2000: maintainer change, since Bruno Haible was busy doing CLisp, Linux I18N, Unicode Support (libiconv), GLibC and several other free software projects

2006/2007: release of CLN 1.2 with support for huge numbers (>4GB each)

CLN Features

- rich set of number classes with unlimited precision

  integers, rational numbers, floats, complex numbers, modular integers, even univariate polynomials

- natural mathematical syntax / type system

  algebraic syntax through operator overloading (`z=x+y` instead of `add(x,y,&z)`)
  natural injections like $\mathbb{Z} \longrightarrow \mathbb{Q}$ modeled with types

- speed efficiency

  C++ compiles to good machine code, usage of assembler for critical parts and common CPUs ('i386', 'x86_64', 'alpha', . . .), asymptotically ideal algorithms (Schönhage-Strassen multiplication, binary splitting, etc.)

- memory efficiency

  representation of small numbers as immediate values instead of as pointers to heap allocated storage
  object sharing: $x+0$ returns x without copying it, etc.

CLN Type System

Natural injections in an OO environment: $\mathbb{Z} \longrightarrow \mathbb{Q}$, $\mathbb{Q} \longrightarrow \mathbb{R}$, $\mathbb{R} \longrightarrow \mathbb{C}$, etc.

CLN types for those fields:

Integers $\mathbb{Z}$: `cl_I`

Rationals $\mathbb{Q}$: `cl_RA`

Reals $\mathbb{R}$: `cl_R`

Complex numbers $\mathbb{C}$: `cl_N`

CLN Type System

Natural injections in an OO environment: $\mathbb{Z} \longrightarrow \mathbb{Q}$, $\mathbb{Q} \longrightarrow \mathbb{R}$, $\mathbb{R} \longrightarrow \mathbb{C}$, etc.

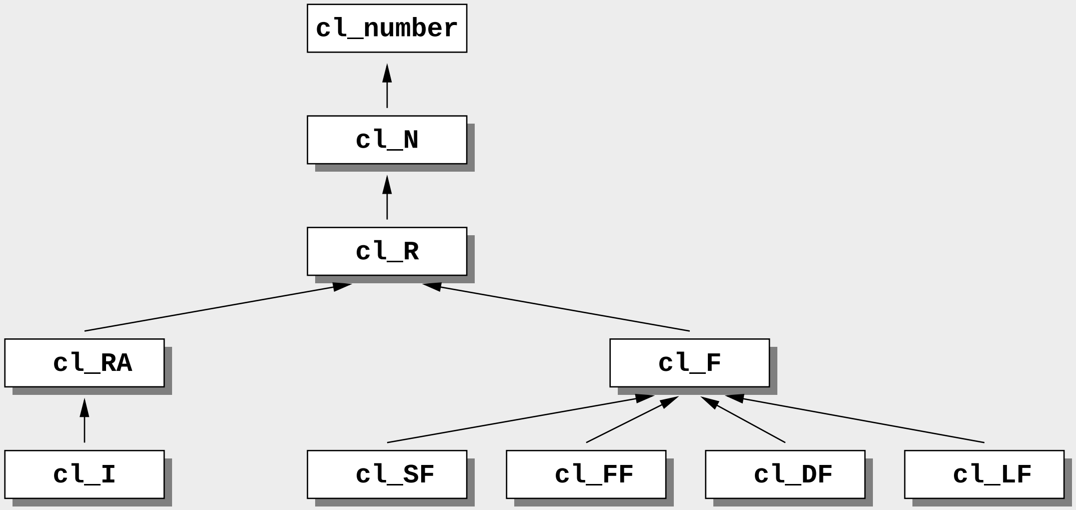
CLN types for those fields:

Integers $\mathbb{Z}$: `cl_I`

Rationals $\mathbb{Q}$: `cl_RA`

Reals $\mathbb{R}$: `cl_R`

Complex numbers $\mathbb{C}$: `cl_N`



CLN Type System

Natural injections in an OO environment: $\mathbb{Z} \to \mathbb{Q}$, $\mathbb{Q} \to \mathbb{R}$, $\mathbb{R} \to \mathbb{C}$, etc.

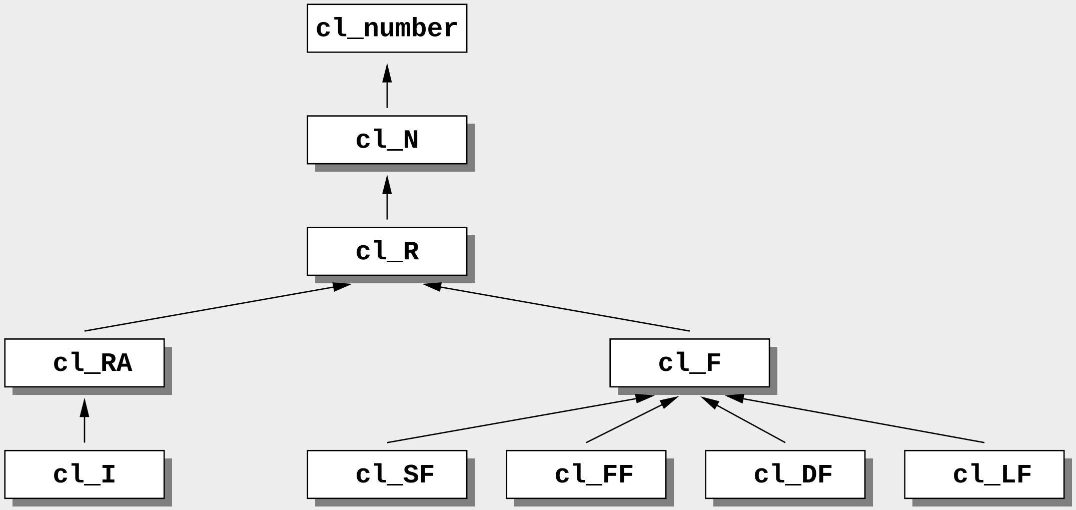
CLN types for those fields:

Integers $\mathbb{Z}$: `cl_I`

Rationals $\mathbb{Q}$: `cl_RA`

Reals $\mathbb{R}$: `cl_R`

Complex numbers $\mathbb{C}$: `cl_N`



Short-Floats `cl_SF`: sign, 17 mantissa bits, 8 exponent bits

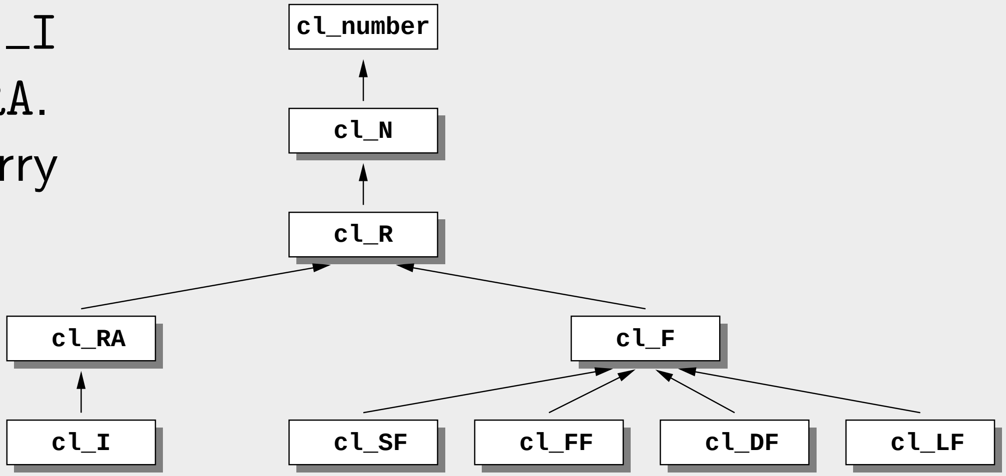
Single-Floats `cl_FF`: sign, 24 mantissa bits, 8 exponent bits
(IEEE 754 single-precision floating point number type)

Double-Floats `cl_DF`: sign, 53 mantissa bits, 11 exponent bits
(IEEE 754 double-precision floating point number type)

Long-Floats: `cl_LF` sign, arbitrary number of mantissa bits, 32 (or 64) exponent bits

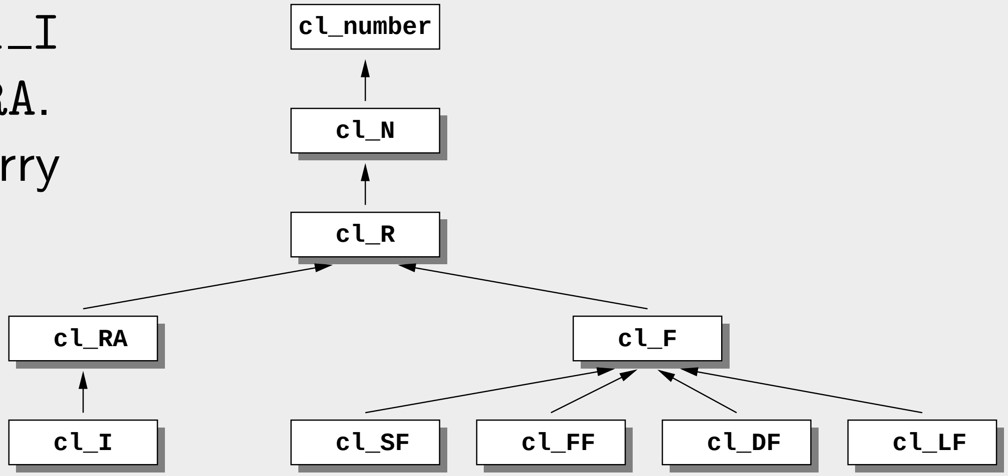
Implementation Aspects: Implementing `class cl_I : public cl_RA`

Problem: Mathematically, `cl_I` must be a specialization of `cl_RA`. But doesn't a rational number carry more data than an integer? Isn't this anti-OO?



Implementation Aspects: Implementing `class cl_I : public cl_RA`

Problem: Mathematically, `cl_I` must be a specialization of `cl_RA`. But doesn't a rational number carry more data than an integer? Isn't this anti-OO?



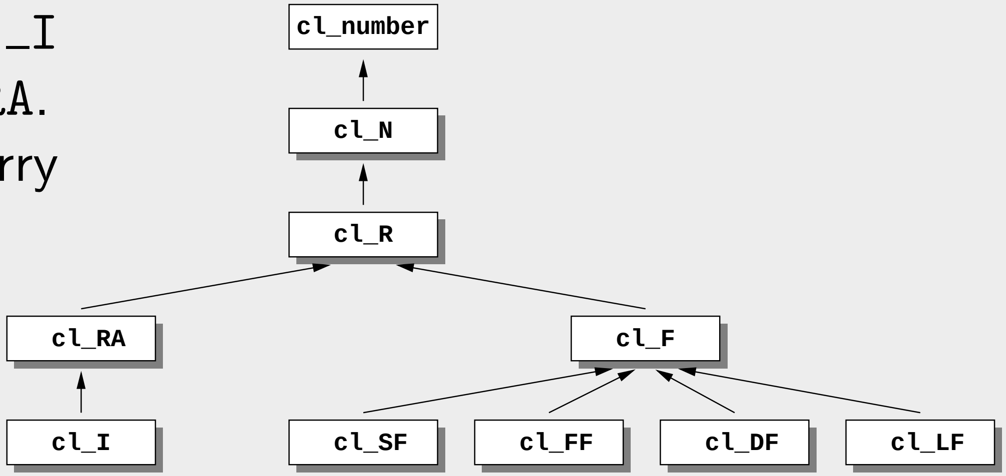
cl_number
cl_N
cl_R
cl_RA
cl_F
cl_I
cl_SF
cl_FF
cl_DF
cl_LF

Solution: implementation follows the „bridge“ design pattern:

```
class cl_number {
    void* pointer_to_hidden_implementation;
    // no other data fields
};
...
class cl_RA : public cl_R {
    // no new data fields...
};
class cl_I : public cl_RA {
    // no new data fields...
};
```

Implementation Aspects: Implementing `class cl_I : public cl_RA`

Problem: Mathematically, `cl_I` must be a specialization of `cl_RA`. But doesn't a rational number carry more data than an integer? Isn't this anti-OO?



Solution: implementation follows the „bridge" design pattern:

```
class cl_number {
    void* pointer_to_hidden_implementation;
    // no other data fields
};
...
class cl_RA : public cl_R {
    // no new data fields...
};
class cl_I : public cl_RA {
    // no new data fields...
};
```

Consequence: `sizeof(cl_number)` $= \cdots =$ `sizeof(cl_I)` $= 4$ (or 8)

Implementation Aspects: Opportunities of the Bridge Design Pattern

So, a user-accessible object is really just a pointer disguised as a type.

- Intrusive reference counting of heap-allocated memory:
  efficient, non-interruptive garbage collection
- Declare `x`, `y`, `z` of type `cl_RA` (i.e. $\in \mathbb{Q}$)
  let `x` $= 3/2$, `y` $= 1/2$, and `z` $=$ `x+y`
  integrality test can be implemented efficiently. User code:
  ```
  if (instanceof(z, cl_I_ring)) {
      // will be true, even though typeof(z) is cl_RA!
  ```
- Object sharing: $x+0$ returns x without copying it, etc.
- Small integers and short floats are *immediate*, not heap allocated

Implementation Aspects: Fixnums and Bignums

Implementation Aspects: Fixnums and Bignums

Chunks of memory on the free store are always aligned.

Return values of `malloc(3)` and friends are multiples of 4 (or 8).

On a typical 32-bit system, such an address is:

$$b_{00}b_{01}b_{02}b_{03}b_{04}b_{05}b_{06}b_{07}\ b_{08}b_{09}b_{10}b_{11}b_{12}b_{13}b_{14}b_{15}\ b_{16}b_{17}b_{18}b_{19}b_{20}b_{21}b_{22}b_{23}\ b_{24}b_{25}b_{26}b_{27}b_{28}b_{29}0\ 0$$

Implementation Aspects: Fixnums and Bignums

Chunks of memory on the free store are always aligned.
Return values of `malloc(3)` and friends are multiples of 4 (or 8).
On a typical 32-bit system, such an address is:

$$b_{00}b_{01}b_{02}b_{03}b_{04}b_{05}b_{06}b_{07}\ b_{08}b_{09}b_{10}b_{11}b_{12}b_{13}b_{14}b_{15}\ b_{16}b_{17}b_{18}b_{19}b_{20}b_{21}b_{22}b_{23}\ b_{24}b_{25}b_{26}b_{27}b_{28}b_{29}0\ 0$$

That leaves 2 (or 3) unused bits which are always zero.
If any of these bits is non-zero, let's interpret the 2 (or 3) bits as a „*type-tag*“ and the remaining bits b_{00}-b_{29} as immediate data.

Implementation Aspects: Fixnums and Bignums

Chunks of memory on the free store are always aligned.
Return values of `malloc(3)` and friends are multiples of 4 (or 8).
On a typical 32-bit system, such an address is:

$$b_{00}b_{01}b_{02}b_{03}b_{04}b_{05}b_{06}b_{07}\ b_{08}b_{09}b_{10}b_{11}b_{12}b_{13}b_{14}b_{15}\ b_{16}b_{17}b_{18}b_{19}b_{20}b_{21}b_{22}b_{23}\ b_{24}b_{25}b_{26}b_{27}b_{28}b_{29}0\ 0$$

That leaves 2 (or 3) unused bits which are always zero.
If any of these bits is non-zero, let's interpret the 2 (or 3) bits as a „*type-tag*“ and the remaining bits b_{00}-b_{29} as immediate data.

Two examples:

- $b_{00} \dots b_{29}$ represent a signed integer in two's complement notation and constant tags $b_{30} = 0$, $b_{31} = 1$ (immediate integers $-2^{29} \dots 2^{29}-1$)
- b_{00} represents a sign, $b_{01} \dots b_{08}$ an exponent, $b_{09} \dots b_{24}$ a mantissa, and constant tags $b_{30} = 1$, $b_{31} = 0$ (immediate short float type `cl_SF`)

Implementation Aspects: Fixnums and Bignums

Chunks of memory on the free store are always aligned.
Return values of `malloc(3)` and friends are multiples of 4 (or 8).
On a typical 32-bit system, such an address is:

$$b_{00}b_{01}b_{02}b_{03}b_{04}b_{05}b_{06}b_{07}\ b_{08}b_{09}b_{10}b_{11}b_{12}b_{13}b_{14}b_{15}\ b_{16}b_{17}b_{18}b_{19}b_{20}b_{21}b_{22}b_{23}\ b_{24}b_{25}b_{26}b_{27}b_{28}b_{29}0\ 0$$

That leaves 2 (or 3) unused bits which are always zero.
If any of these bits is non-zero, let's interpret the 2 (or 3) bits as a „*type-tag*“ and the remaining bits b_{00}-b_{29} as immediate data.

Two examples:

- $b_{00}\ldots b_{29}$ represent a signed integer in two's complement notation and constant tags $b_{30}=0$, $b_{31}=1$ (immediate integers $-2^{29}\ldots 2^{29}-1$)
- b_{00} represents a sign, $b_{01}\ldots b_{08}$ an exponent, $b_{09}\ldots b_{24}$ a mantissa, and constant tags $b_{30}=1$, $b_{31}=0$ (immediate short float type `cl_SF`)

$\Rightarrow$ no heap allocation for small values $\Rightarrow$ efficiency
all this is completely transparent for the user of the library

Implementation Aspects: Fixnums and Bignums

Chunks of memory on the free store are always aligned.
Return values of `malloc(3)` and friends are multiples of 4 (or 8).
On a typical 32-bit system, such an address is:

$$b_{00}b_{01}b_{02}b_{03}b_{04}b_{05}b_{06}b_{07}\ b_{08}b_{09}b_{10}b_{11}b_{12}b_{13}b_{14}b_{15}\ b_{16}b_{17}b_{18}b_{19}b_{20}b_{21}b_{22}b_{23}\ b_{24}b_{25}b_{26}b_{27}b_{28}b_{29}0\ 0$$

That leaves 2 (or 3) unused bits which are always zero.
If any of these bits is non-zero, let's interpret the 2 (or 3) bits as a „*type-tag*“ and the remaining bits b_{00}-b_{29} as immediate data.

Two examples:

- $b_{00} \ldots b_{29}$ represent a signed integer in two's complement notation and constant tags $b_{30} = 0$, $b_{31} = 1$ (immediate integers $-2^{29} \ldots 2^{29}-1$)
- b_{00} represents a sign, $b_{01} \ldots b_{08}$ an exponent, $b_{09} \ldots b_{24}$ a mantissa, and constant tags $b_{30} = 1$, $b_{31} = 0$ (immediate short float type `cl_SF`)

⇒ no heap allocation for small values ⇒ efficiency
all this is completely transparent for the user of the library

Small CLN Example

For the largest known Mersenne prime p, compute $(2^p - 1)2^{p-1}$.
This is the largest known perfect number:

```
#include <iostream>
#include <cln/cln.h>
using namespace std;
using namespace cln;

int main()
{
    int p = 30402457;
    cl_I x = ((cl_I(1) << p) - 1) << (p-1);
    cout << x << endl;
}
```

⇒ printing 18304103 decimal digits takes ca. 1 minute

Finally. . .

CLN has been around and stable for a very long time
licensed under GPL and available from `http://www.ginac.de/CLN/`
pre-packaged in debian, SuSE and FreeBSD
current focus is stability and evolution in small steps
Some projects using it:

- GiNaC `http://www.ginac.de/`
  symbolic system in C++ for use within C++
- Qalculate! `http://qalculate.sourceforge.net/`
  GUI desktop calculator on steroids
- RPN-Calculator-Py `http://sourceforge.net/projects/calcrpnpy/`
  reverse polish notation interpreter for use as an interactive calculator in conjunction with the Python interactive interpreter
- gTybalt `http://wwwthep.physik.uni-mainz.de/~stefanw/gtybalt/`
  combination of several C++ packages (CLN, GiNaC, NTL) under CERN's Root framework