CNCL

Communication Networks Class Library
Edition 1.10, $Date: 1998-01-24 18:40:45+01 $

by M. Junius, M. Steppler, M. Büter, D. Pesch, and others

1 Introduction to the CNCL

CNCL is a C++ library created at Communication Networks, Aachen University of Technology, Germany.

The main objective of this class library is to provide a common base for all C++ applications created at Communication Networks. Therefore, CNCL is both a class library featuring generic C++ classes as well as a simulation library with strong points in random number generation, statistics, and event driven simulation.

All globally visible classes and type definitions feature a "`CN`" prepended to their name to avoid collision with other class libraries, e.g. GNU libg++.

1.1 CNCL Class Hierarchy

`CNCL`	CNCL Static Members and Functions
`  CNObject`	Root of the CNCL Hierarchy
`    CNClass`	Class Description
`    CNParam`	Abstract Parameter Base Class
`    CNRNG`	Abstract Random Number Generator Base Class
`      CNACG`	Additive RNG
`      CNFiboG`	Fibonacci RNG
`      CNFileG`	Data File RNG
`      CNLCG`	Linear Congruence RNG
`      CNMLCG`	Multiple Linear Congruence RNG
`      CNTausG`	Tausworth RNG
`    CNRndInt`	Random Integers
`    CNRandom`	Abstract Random Distribution Base Class
`      CNBeta`	Beta Distribution
`      CNBinomial`	Binomial Distribution
`      CNDeterm`	Deterministic Distribution
`      CNDiracTab`	Distribution from Table
`      CNDiscUniform`	Discrete Uniform Distribution
`      CNErlang`	Erlang-k Distribution
`      CNGeometric`	Geometric Distributed Random Numbers
`      CNHyperExp`	HyperExp Distributed Random Numbers
`      CNHyperGeom`	HyperGeom Distributed Random Numbers
`      CNMDeterm`	Random Mix of Deterministic Distributions
`      CNNegExp`	Negative Exponential Distribution
`      CNNormal`	Normal Distribution
`        CNLogNormal`	LogNormal Distribution
`        CNRayleigh`	Rayleigh Distribution
`        CNRice`	Rice Distribution
`      CNPoisson`	Poisson Distribution
`      CNRandomMix`	Mix of Several `CNRandom` Distributions
`      CNTab`	Distribution from Table
`        CNInterTab`	Distribution from Table (Interpolated Values)
`      CNUniform`	Uniform Distribution
`      CNWeibull`	Weibull Distribution
`    CNStatistics`	Abstract Statistics Evaluation Base Class
`      CNMoments`	Evaluation of (Weighted) Moments
`      CNMomentsTime`	Evaluation of Time-Weighted Moments
`      CNConfidence`	Evaluation of Moments with Confidence Intervals
`      CNLRE`	LRE Base Class
`        CNLREF`	Evaluation by LRE F(x)
`        CNLREG`	Evaluation by LRE G(x)
`      CNDLREF`	Evaluation by Discrete LRE F(x)
`      CNDLREG`	Evaluation by Discrete LRE G(x)
`      CNBatchMeans`	Evaluation by Batch Means
`    CNAVLTree`	AVL balanced binary search tree
`    CNAVLNode`	Node of AVL tree

`CNSLList`	Single Linked List of Objects
`CNDLList`	Doubly Linked List of Objects
`CNSLObject`	Node of Single Linked List
`CNDLObject`	Node of DoubleyLinked List
`CNSLIterator`	Iterator of Single Linked List
`CNDLIterator`	Iterator of Doubly Linked List
`CNStack`	Stack
`CNQueue`	Abstract Queue Base Class
`CNQueueFIFO`	FIFO Queue
`CNQueueLIFO`	LIFO Queue
`CNQueueRandom`	Random Queue
`CNQueueSPT`	SPT Queue
`CNPrioQueueFIFO`	Priority Queue
`CNSink`	Sink
`CNJob`	Standard Job for Queues

`CNDLObject`	
`CNEvent`	Generic Event
`CNNamed`	
`CNEventHandler`	Abstract Base Class for Event Handlers
`CNEventExploder`	Send Events to multiple EventHandlers
`CNEventList`	List of Events
`CNEventBaseSched`	
`CNEventScheduler`	Event Scheduler
`CNEventHeapSched`	Event Scheduler using a heap
`CNSimTime`	Simulation Time

`CNArray`	Abstract Base class of 1-dimensional Arrays
`CNArrayObject`	Array of Pointer to `CNObject`
`CNArrayChar`	Array of `Char`
`CNArrayDouble`	Array of `Double`
`CNArrayFloat`	Array of `Float`
`CNArrayInt`	Array of `Int`
`CNArrayLong`	Array of `Long`
`CNArray2`	Abstract Base class of 2-dimensional Arrays
`CNArray2Object`	2-dimensional Array of Pointer to `CNObject`
`CNArray2Char`	2-dimensional Array of `Char`
`CNArray2Double`	2-dimensional Array of `Double`
`CNArray2Float`	2-dimensional Array of `Float`
`CNArray2Int`	2-dimensional Array of `Int`
`CNArray2Long`	2-dimensional Array of `Long`
`CNArray3`	Abstract Base class of 3-dimensional Arrays
`CNArray3Object`	3-dimensional Array of Pointer to `CNObject`
`CNArray3Char`	3-dimensional Array of `Char`
`CNArray3Double`	3-dimensional Array of `Double`
`CNArray3Float`	3-dimensional Array of `Float`
`CNArray3Int`	3-dimensional Array of `Int`
`CNArray3Long`	3-dimensional Array of `Long`

`CNKey`	Abstract Base Class for Keys
`CNKeyString`	String Key

`CNKeyInt`	Integer Key
`CNHashTable`	Abstract Hash Table Base Class
`CNHashStatic`	Hash Tables with Static Capacity
`CNHashDynamic`	Hash Tables with Dynamic Capacity
`CNHashIterator`	Sequential Iterator for Hash Tables
`CNManager`	Object Management Frontend
`CNCoord`	2-Dimensional Coordinates
`CNICoord`	2-Dimensional Integer Coordinates
`CNString`	Character String
`CNFormInt`	Integer as CNStrings
`CNFormFloat`	Doubles as CNStrings
`CNNamed`	Object with Name
`CNIniFile`	.ini-style config file
`CNInt`	Integer derived from CNObject
`CNDouble`	Double derived from CNObject
`CNGetOpt`	Interface to GNU getopt()
`CNRef`	Base class for classes with reference counting
`CNObject`	
`CNRefObj`	CNObject with reference counting
`CNNamed`	
`CNRefNamed`	CNNamed with reference counting
`CNPtr`	Intelligent pointer to CNRefObjs
`CNPipe`	UNIX Pipe
`CNSelect`	UNIX Select Interface
`CNNamed`	
`EZD`	Base Class for EZD Graphic Objects
`EZDDrawing`	Interface to EZD Drawings
`EZDPushButton`	Interface to EZD Push-Button
`EZDWindow`	Interface to EZD Windows
`EZDDiagWin`	Extra window with x-y diagram
`EZDTextWin`	EZD window for easy text display
`EZDObject`	Interface to EZD Object
`EZDDiag`	x-y diagram as an EZDObject
`EZDBlock`	Block with small rectangles for bit display
`EZDPopUp`	Interface to EZD popup menu
`EZDQueue`	Graphical Representation of a Queue
`EZDServer`	Graphical Representation of a Server
`EZDText`	EZD Object with Text
`EZDTimer`	Graphical Representation of a Timer
`CNFClause`	Clause of a fuzzy rule
`CNFRule`	Fuzzy rule
`CNNamed`	
`CNFVar`	Fuzzy variable
`CNFRuleBase`	Rule base and fuzzy inference engine
`CNFSet`	Fuzzy set abstract base class
`CNFSetArray`	Fuzzy set based on array with membership values
`CNFSetLR`	Fuzzy set with L and R functions

CNFSetTrapez	Fuzzy set with trapezium function
CNFSetTriangle	Fuzzy set with triangle function
CNFNumTriangle	Fuzzy numbers (triangle)
CNReaderTbl	Table for adress of reader-function
CNPIO	persistent stream Object IO-formatting
CNObject	
CNPstream	abstract base class for persistent stream class
CNPiostream	persistent iostream format
CNPObjectID	ID-Managment for persistent classes
CNPInt	class persistent CNInt
CNP<type>	Other persistent classes

1.2 Common CNCL Member Functions

CNCL requires that all classes have a common set of member functions available. These functions provide runtime type checking and type information, creation of objects via class descriptions, and safe type casts.

The common member functions are:

CNClassDesc *CLASS*::class_desc() const;
: This function returns a pointer to the class description object, which must be available for every class in the CNCL hierarchy. This pointer is used for runtime type information.

bool *CLASS*::is_a(CNClassDesc desc) const;
: This function allows runtime type checking. It returns TRUE if the queried object is type compatible with class desc.

void *CLASS*::print(ostream &strm = cout) const;

void *CLASS*::dump(ostream &strm = cout) const;
: These functions output the object to the given stream strm. The function print() is defined in greater detail in the derived classes. The function dump() is intended for debug purposes.

The functions above are defined as pure virtual functions in the top-level class CNObject. They are required in every derived class.

Furthermore the following static member functions are required to allow object creation via the class description and safe type casts:

*CLASS** *CLASS*::cast_from_object(CNObject *obj);
: This function does a safe (*CLASS* *) type cast. It checks on type compatibility between the object passed with the obj pointer and *CLASS*. If this is a not true, an error message is printed and the program terminates.

: The type checking may be disabled by defining the preprocessor macro NO_TYPE_CHECK, e.g. by supplying -DNO_TYPE_CHECK on the compiler's command line.

: Example:

```
XYZ *px;                // CNClass XYZ derived from CNObject
ABC *pa;                // CNClass ABC derived from CNObject
DEF *pd;                // CNClass DEF derived from ABC
CNObject *po;

po = new DEF;           // Type compatible (C++ standard)
pd = (DEF *)po;         // The traditional way
pd = DEF::cast_from_object(po); // The CNCL way

pa = new DEF;
pd = DEF::cast_from_object(pa); // o.k.
px = XYZ::cast_from_object(pa); // Error
```

`CNObject *`*CLASS*`::new_object(CNParam *param = NIL);`
: This function creates an object of type *CLASS*, optionally passing a pointer to a parameter object to the constructor.

  It is used by the class description `CNClass` to create new objects.

Every class in CNCL requires a class description object and a pointer constant pointing to this class description. Following the CNCL convention, the description object is named *CLASS*`_desc` and the pointer is named `CN_`*CLASS*.

Example:

```
// Describing object for class XYZ
static CNClass XYZ_desc("XYZ", "$Revision: 0.45 $", XYZ::new_object);

// "Type" for type checking functions
CNClassDesc CN_XYZ = &XYZ_desc;
```

1.3 `CNgenclass` Script

To generate new classes for the CNCL hierarchy in a convenient way, the utility `CNgenclass` is provided. It generates a class framework with all the functions required by CNCL.

Usage:

`CNgenclass` *name base*

The required parameters are the name of the class to be created and the name of the base class. The result are two files in the current directory: *name*`.h` (class header file) and *name*`.c` (class implementation file).

Example:

```
CNgenclass MyClass CNObject
```

creates the files `MyClass.h` and `MyClass.c`. Please note that a leading "`CN`" is removed from the file names.

1.4 `minmax` header file

This header file is copied from the GNU libg++ library. Its min(), max() definitions are quite useful and appear several times at CNCL. As minmax is not included with all C++ libraries, this header file is added to CNCL.

The `inline` min() and the max() functions are declared for (un-)unsigned char, (un-)signed short, (un-)signed int, (un-)signed long, float and double.

2 The Basic Classes of the CNCL Hierarchy

The following classes constitute the basics of CNCL. They provide the framework for runtime type checking, class descriptions, object management, and error handling.

2.1 CNCL — CNCL Static Members and Functions

SYNOPSIS

```
#include <CNCL/CNCL.h>
```

TYPE

None

BASE CLASSES

None

DERIVED CLASSES

CNObject

RELATED CLASSES

None

DESCRIPTION

The `CNCL` class contains only static members and functions for the class library's parameters and error handling. All classes in the CNCL hierarchy can directly access the static member functions, other code can call them via `CNCL::`*func*`()`.

ERROR HANDLING

The class `CNCL` provides common functionality for error handling. This is used by all classes to issue an error message or warning and terminate the program, if desired. The CNCL library also installs a `matherr()` handler using `CNCL::error()` for displaying an appropiate error message.

```
enum ErrorType
{
    err_fatal, err_abort, err_warning, err_ignore, err_default, err_info
};
```

The setting of the CNCL error handling:

`err_abort`

`err_fatal`
: Print error message, then terminate the program. (To do so, CNCL calls the exit handler installed with `set_exit_handler()`.) `err_abort` is like `err_fatal` but calls `abort()` for termination resulting in a core dump.

`err_warning`
: Print error message with "warning" prefix.

`err_ignore`
: No error message.

`err_ignore`
: No error, just the message for info.

`err_default`
: The default setting of the CNCL error handling. The default is `err_fatal` (may be set with `set_error()`).

The following member functions of `CNCL` can be used to manipulate the error handling.

`static ErrorType get_error();`
: Returns the current error handling type.

`static ErrorType set_error(ErrorType err);`
: Sets the error handling type to `err` and returns the previous value.

`static void set_exit_handler(void (*func)());`
: Installs a function to be called on fatal errors. The default function will print a message and terminate the program by calling `exit()` (`err_fatal`) or `abort()` (`err_abort`).

`static void default_exit_handler()`
: The CNCL default handler called on fatal errors.

ERROR MESSAGES

The following member function can be used to output error messages and/or terminate the program. Each of the functions accepts up to six `const char *` arguments, if the first one of these is `NIL`, then the default string (`"CNCL error: "`, `"CNCL warning: "`, `"CNCL: "` for `err_error/err_abort`, `err_warning`, `err_info`, respectively) is prepended to the output.

`static void error(const char *msg1 = NIL, ...)`
: Output error message, default error handling. Up to 6 different `char *msg`'s can be added.

`void error(ErrorType err, const char *msg1 = NIL, ...)`
: Output error message, error handling as specified by `err`.Up to 6 different `char *msg`'s can be added.

`static void fatal(const char *msg1 = NIL, ...)`
: Output error message, fatal error handling.Up to 6 different `char *msg`'s can be added.

`static void warning(const char *msg1 = NIL, ...)`
: Output error message, warning error handling.Up to 6 different `char *msg`'s can be added.

`static void info(const char *msg1 = NIL, ...)`
: Output message.Up to 6 different `char *msg`'s can be added.

`static ostream& msg()`
: Returns a reference to an output stream. Output to this stream will be appended to the next `error()`, `fatal()`, `warning()`, `info()` message. This is actually a `strstream` with a maximum capacity of `CNCL::STR_BUF_SIZE`.

UTILITIES

Outside the `CNCL` class the following constants and types are defined:

`NIL` The null pointer defined as `0`.

`TRUE` The boolean value true defined as `1`.

`FALSE` The boolean value false defined as `0`.

`bool` A boolean data type, actually `typedef int bool`.

2.2 CNObject — Root of the CNCL Hierarchy

SYNOPSIS

`#include <CNCL/Object.h>`

TYPE

`CN_OBJECT`

BASE CLASSES

CNCL

DERIVED CLASSES

CNClass, CNParam, ...

RELATED CLASSES

CNClass, CNParam

DESCRIPTION

`CNObject` is the actual base of the CNCL inheritance tree. It must be supported by all derived classes.

`virtual CNClassDesc class_desc() const;`
: Returns the class description (pointer to instance of CNClass) for runtime type information.

`virtual bool is_a(CNClassDesc desc) const;`
: Returns TRUE if the queried object is of type `desc`, else FALSE.

`virtual void print(ostream &strm = cout) const = 0;`
: Output object to stream.

`virtual void dump(ostream &strm = cout) const = 0;`
: Output object to stream for debug purpose.

`virtual int store_on(CNPstream &);`
`virtual int storer(CNPstream &);`
: These functions are support functions for persistent objects, so they are not implemented in all derived classes. They are yielding a warning if they are called for a non-persistent object.

UTILITIES

`Object.h` defines the following operators for easily writing objects to streams:

`ostream &operator << (ostream &strm, const CNObject &obj);`

`ostream &operator << (ostream &strm, const CNObject *obj);`

Write object to stream using the `print()` member function. It is safe to output a null pointer `obj`, in this case "`(NIL)`" is printed.

2.3 CNClass — Class Description

SYNOPSIS

```
#include <CNCL/Class.h>
```

TYPE

`CN_CLASS`

BASE CLASSES

CNObject

DERIVED CLASSES

None

RELATED CLASSES

CNObject

DESCRIPTION

`CNClass` is used for the type description objects. For each class in the CNCL hierarchy there is a corresponding object of type `CNClass`. A pointer to this object is used for the CNCL runtime type information.

Constructors:

`CNClass(char *name, char *version, CNObject *(*func)(CNParam *p));`
: The constructor of `CNClass` takes three arguments: the class' name, the class' version and a pointer to the static `CLASS::new_object()` member function of classes that provide this functionality. The function pointer may be `NIL` for classes that do not provide this, such as abstract base classes.

In addition to the member functions required by CNCL, `CNClass` provides:

`const char *name() const;`

`const char *get_name() const;`
Returns the class name.

`const char *version() const;`

`const char *get_version() const;`
Returns the class version.

`CNObject *new_object(CNParam *param = NIL) const;`

`CNObject *new_object(Param &param) const;`
Creates new objects via the corresponding class' static member function `new_object()`. The `param` object is used to pass optional arguments to the constructor.

`static CNClass *cast_from_object(CNObject *obj);`
Safes `(CNClass *)` type cast.

2.4 CNParam — Abstract Parameter Base Class

SYNOPSIS

```
#include <CNCL/Param.h>
```

TYPE

`CN_PARAM`

BASE CLASSES

CNObject

DERIVED CLASSES

RELATED CLASSES

CNClass

DESCRIPTION

`CNParam` is the abstract base for parameter classes used to pass constructor parameters to objects in a general way. This will be used in the forthcoming CNCL object management.

3 Random Numbers

CNCL provides a variety of random number generators, ranging from the very simple linear congruence generator to more sophisticated ones. Also included is a random number generator which reads random data from a file. The generator classes are:

CNLCG - Linear Congruential Generator

CNMLCG - Multiplicative Linear Congruential Generator

CNACG - Additive Congruential Generator

CNFiboG - Lagged Fibonacci Generator

CNTausG - Tausworthe Generator

CNFileG - Data file random number generator

Random numbers are a crucial base of every simulation; most of the pseudo random number generators included in CNCL have their faults and shortcomings. The base random number generators are used by the random distribution classes to generate random numbers with the desired distribution. For futher information refer to the description of the generator you are interested in. Here you can learn something about seed selection in general. The cited references, also the references of the generator-sections, are collected here.

Seed Selection

Due to Lewis and Orav [4] there are some aspects to consider when selecting the seeds for the generators: Ordinarily, the seed value used to initialize a random number generator should not affect the results of the simulation. However, a wrong combination of a seed and a random generator may lead to erroneous conclusions. If the generator has a full period and only one random variable is required, any seed value is as good as any other. However, considerable care is required in selecting seeds for simulations requiring random numbers for more than one variable. Such simulations are called multistream simulations. Some guidelines to follow:

1. If possible, use the same base generator with one seed for all random variables, i.e. successive values of different random variables use successive values of only one base generator. Thus the problem of overlapping streams (see 4.) is avoided.
2. Do not use zero. It would make a multiplicative LCG or a Tausworthe generator stick at zero. According to Knuth [1], the seed for an LCG and its modulus m should be relative prime.
3. Avoid even values. Even values are often as good as odd values. In fact for full period generators, all nonzero seed values are equally good.
4. If distinct streams are needed (see 1.), use nonoverlapping streams. Each stream requires a separate seed. If the two seeds are such that the two streams overlap, there will be a correlation between the streams, and the resulting sequences will not be independent.
5. Reuse seeds in successive replications. If a simulation experiment is replicated several times, the random-number stream need not to be reinitialized, and the seeds left over from the previous replication can continue to be used.
6. Do not use random seeds. Analyst often use random-seed values such as the time of the day. This causes two problems: first, the simulation cannot be reproduced and, second, it is not possible to guarantee that the multiple streams will not overlap. Random-seed selection is therefore not recommended. In particular, do not use successive random numbers obtained from the generator as seeds.

Memory Requirements

LCG	1 longword = 4 bytes
MLCG	2 longwords = 8 bytes
ACG	depends on table size, with default size (55) = 55 + 256 longwords = 1244 bytes
FiboG	102 longwords = 408 bytes
TausG	5 longwords = 20 bytes

For each instance of the generator (on 32bit machines)

Period Length

LCG	2^31- 2
MLCG	2.3 * 10^18
ACG	depends on table size
FiboG	(2^32-1) * 2^96
TausG	-

Evaluation

```
      | Randomness | Period    | Performance | Memory
      |            | Length    |             | Requirements
------|------------|-----------|-------------|---------------
LCG   |    +       |   +       |   +++++     |   +++++
MLCG  |    ++      |   ++      |   +         |   ++++
ACG   |    +++     |   1)      |   ++        |   +
FiboG |    +++++   |   +++++   |   +++       |   ++
TausG |    ++      |   -       |   ++++      |   +++
------------------------------------------------------------
1) depends on table size
```

Performance evaluation took place on Sun workstations, results for other processors may differ.

Implementation Details

If you are interested in implementation details of the random number generators Press, Vetterling, Teukolsky and Flannery [14], especially chapter 7, may be an interesting first reading. (e.g. Schrage's algorithm is described and a quick and dirty way to convert two longs into one double distributed between zero and 1).

References

[1] Knuth, D.E. *Seminumerical Algorithms.* 2nd ed., vol.2 of The Art of Computer Programming, Chapter 3, Addison-Wesley, Reading MA, 1981.

[2] Park, S.K., and Miller, K.W., Communications of the ACM, vol.31, pp. 1192-1201, 1988.

[3] Schrage, L., ACM Transactions on Mathematical Software, vol.5, pp. 132-138, 1979.

[4] Lewis, P.A.W, and Orav, E.J. *Simulation Methodology forStatisticians, Operations Analysts, and Engineers.* vol.1 pp. 88-97, Wadsworth & Brooks/Cole, 1989.

[5] Fishmann, G. and Moore, L. *An Exhaustive Analysis of Multiplicative Congruential Random Number Generators with Modulus 2^31-1.* SIAM Journal of Scientific and Statistical Computing, 7(1), pp. 24-45, 1985.

[6] L'Ecuyer, P. *Efficient and Portable Combined Random Number Generators.* Communications of the ACM, vol. 31 (6), pp. 742-774, 1988.

[7] Jain, R. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Chapter 26, John Wiley & Sons, Inc., 1991.

[8] Lehmer, D.H.: *Mathematical Methods in Large-Scale Computing Units.* Ann. Comput. Lab. Harvard Univ., 26, pp. 141-146, 1951.

[9] Marsaglia, G., Zaman, A. and Tsang, W.-W., Stat. Prob. Lett. 9, 35, 1990.

[10] James, F. *A review of pseudorandom number generators.* Computer Physics Communication 60, pp. 329-344, North Holland, 1990.

[11] Richter, M. *Ein Rauschgenerator zur Gewinnung von quasi-idealen Zufallszahlen fuer die stochastische Simulation.* PhD thesis, Aachen University of Technology, 1992.

[12] Tausworthe, R. C. *Random Numbers Generated by Linear Recurrence Modulo Two.* Math. Comput. 19, pp. 201-209, 1965.

[13] Law, A.M., Kelton, W.D. *Simulation Modeling & Analysis.* Chapter 7, McGraw Hill, 1991.

[14] Press, W.H., Vetterling, W.T., Teukolsky, S.A., Flannery, B.P. *Numerical Recipes in C - The Art of Scientific Computing.* 2nd ed., Cambridge University Press, 1994.

3.1 CNRNG — Abstract Random Number Generator Base Class

SYNOPSIS

`#include <CNCL/RNG.h>`

TYPE

`CN_RNG`

BASE CLASSES

CNObject

DERIVED CLASSES

CNACG, CNFiboG, CNFileG, CNLCG, CNMLCG, CNTausG

RELATED CLASSES

CNRandom

DESCRIPTION

`CNRNG` is the abstract base class for all CNCL random number generators. It defines the common interface.

Constructors:

`CNRNG();`

`CNRNG(CNParam *param);`
Initializes `CNRNG`.

In addition to the member functions required by CNCL, `CNRNG` provides:

`unsigned long as_long();`
This function returns an unsigned integer in the range 0 ... 2^31-1. It uses the as_long32() function from the actual CNRNG to draw a random number and truncates it down to 31bit.

`virtual unsigned long as_long32() = 0;`
: This function is used to draw a random number from the actual CNRNG (derived from the `CNRNG` class). The result is an unsigned integer in the range 0 ... 2^32-1 provided the class is able to produce 32bit random numbers.

`virtual bool has_long32() = 0;`
: This function tells whether the actual CNRNG is able to produce 32bit integer values or not.

`virtual void reset() = 0;`
: Resets the CNRNG to its initial state.

`float as_float();`

`double as_double();`
: These functions draw a random number in the range 0 ... 1 and return the result as a `float` or `double` value.

`virtual void seed(unsigned long s);`
: This method for all RNG's only draws `s` as_long32() numbers.

3.2 CNACG — Additive RNG

SYNOPSIS

```
#include <CNCL/ACG.h>
```

TYPE

`CN_ACG`

BASE CLASSES

CNRNG

DERIVED CLASSES

None

RELATED CLASSES

CNRandom

DESCRIPTION

`CNACG` is the additive random number generator class. It is a variant of a Linear Congruential Generator (Algorithm M) described in Knuth [1]. The results undergo a permutation controlled by a Fibonacci Additive Congruential Generator to get good independence between samples. This is a very high quality random number generator, although it requires a fair amount of memory for each instance of the generator. The implementation was taken, like the MLCG, from the GNU-library libg++.

The 'CNACG::CNACG' constructor takes two parameters: the seed and the size. The seed is any number to be used as an initial seed. The performance of the generator depends on having a distribution of bits through the seed. If you choose a number in the range of 0 to 31, a seed with more bits is chosen. Other values are deterministically modified to give a better distribution of bits. This provides a good random number generator while still allowing a sequence to be repeated given the same initial seed.

The 'size' parameter determines the size of two tables used in the generator. The first table is used in the Additive Generator; see the algorithm in Knuth for more information. In general, this table is 'size' longwords long. The default value, used in the algorithm in Knuth, gives a table of 220 bytes. The table size affects the period of the generators; smaller values give shorter periods and larger tables give longer periods. The smallest table size is 7 longwords, and the longest is 98 longwords. The ‘size’ parameter also determines the size of the table used for the Linear Congruential Generator. This value is chosen implicitly based on the size of the Additive Congruential Generator table. It is two powers of two larger than the power of two that is larger than ‘size’. For example, if ‘size’ is 7, the ACG table is 7 longwords and the LCG table is 128 longwords. Thus, the default size (55) requires 55 + 256 longwords, or 1244 bytes. The largest table requires 2440 bytes and the smallest table requires 100 bytes. Applications that require a large number of generators or applications that aren’t so fussy about the quality of the generator may elect to use the ’CNMLCG’ generator instead.

This generator has an extremly long period and provides a good independence. Unfortunately, uniformity is not too great. For further details refer to the comments included in the source ’CNACG.c’.

Constructors:

`CNACG(unsigned long seed = 0, int size = 55);`

`CNACG(CNParam *param);`
: Initializes `CNACG`.

In addition to the member functions required by CNCL, `CNACG` provides:

`virtual unsigned long as_long32();`
: Draws a random number. The result is an unsigned integer in the range 0 ... 2^32-1.

`virtual bool has_long32();`
: Returns TRUE because the CNACG is able to produce 32bit integer values.

`virtual void reset();`
: Resets the CNACG to its initial state.

3.3 CNFiboG — Fibonacci RNG

SYNOPSIS

```
#include <CNCL/FiboG.h>
```

TYPE

`CN_FIBOG`

BASE CLASSES

CNRNG

DERIVED CLASSES

None

RELATED CLASSES

CNRandom

DESCRIPTION

`CNFiboG` is the Fibonacci random number generator class. This generator is a lagged fibonacci generator which is a modification based on the algorithm proposed by Marsaglia, Zaman and Tsang [9]. Concerning the seed selection and the amount of memory required James [10] states the following: "A most exceptional property of this generator is the extreme simplicity of generating independently disjoint sequences. The generator must be initialized by giving one 32-bit integer . . ., each value of which gives rise to an independent sequence of sufficient length for an entire calculation. This means that in a collaboration between different physicists, each physicist can be assigned one number between zero and 9999 as the last four decimal digits of the initiator, and he will be assured of not overlapping the sequences of any other, even though he still has about 90000 possibilities for the other digits at his disposal for independent initialization. That is, the program can generate about 900 million different subsequences, each one very long (average length ca. 10^30). On the other hand, there is a small price to pay for the exceptionally long period: The complete specification of the state of the generator at a given point (for example to be able to regenerate a given event in the middle of a calculation) requires one hundred and two full words... This contrasts with the one word (two words) necessary for a MLCG of period ca. 10^9 (10^18)." The implemented version is based on the following algorithm:

$$V_i = (V_{i-97} - V_{i-37}) \quad mod \quad 2^{32}$$
$$C_i = (C_{i-1} - 362,436,069) \quad mod \quad 2^{32}$$
$$D_i = (V_i - C_i) \quad mod \quad 2^{32}$$

The period of this generator is determined by N = (2^32 - 1) * 2^96. For further details and statistical tests of this generator refer to Richter [11].

The main advantage of this method can be seen in a combination of a simple mathematical formula and a period length sufficient enough for physical simulation runs. Nevertheless, it still represents a pseudo random number generator. Thus, a non-ideal correlation can be expected.

Constructors:

`CNFiboG(CNParam *param);`

`CNFiboG(unsigned long init = 54217137);`
: Initializes `CNFiboG` with a 97 elements circular queue and initial seed.

In addition to the member functions required by CNCL, `CNFiboG` provides:

`virtual unsigned long as_long32();`
: Draws a random number. The result is an unsigned integer in the range 0 ... 2^32-1.

`virtual bool has_long32();`
: Returns TRUE because CNFiboG is able to produce 32bit integer values.

`virtual void reset();`
: Resets the CNFiboG to its initial state.

`void seed_internal(unsigned long *ulp);`
: Reinitializes the circular queue and cn with `*ulp`, an array of 98 values.

3.4 CNFileG — Data File RNG

SYNOPSIS

```
#include <CNCL/FileG.h>
```

TYPE

CN_FILEG

BASE CLASSES

CNRNG

DERIVED CLASSES

None

RELATED CLASSES

CNRandom

DESCRIPTION

`CNFileG` is a data file random number generator class. It reads random numbers from a disk file, e.g. data from PURAN2. Thus the quality of this generator class depends on the quality of the data files. Truely random numbers can be generated if a good file is supplied. Here the problem of this class can be seen. A good file must have a sufficient size. Huge memory use and low speed can be expected when using this class.

Constructors:

`CNFileG(char *filename, bool par = FALSE);`

`CNFileG(CNParam *param);`

Initializes `CNFileG` with data file `filename` and sets parity check if required, e.g. parity check for PURAN2.

In addition to the member functions required by CNCL, `CNFileG` provides:

`virtual unsigned long as_long32();`
: Draws a random number. The result is an unsigned integer in the range 0 ... 2^32-1.

`virtual bool has_long32();`
: Returns TRUE because the CNFileG is able to produce 32bit integer values.

`virtual void reset();`
: Resets the CNFileG to its initial state.

`void newfile(char *filename, bool par = FALSE);`
: Opens a new file for reading random number data and indicates if data requires parity check.

`unsigned int wrong_parity();`
: Gets the number of wrong parity checks while reading data from a file, e.g. data from PURAN2.

3.5 CNLCG — Linear Congruence RNG

SYNOPSIS

```
#include <CNCL/LCG.h>
```

TYPE

`CN_LCG`

BASE CLASSES

CNRNG

DERIVED CLASSES

None

RELATED CLASSES

CNRandom

DESCRIPTION

`CNLCG` is a linear congruence random number generator class. The linear congruential generator is based on the formula

$$X_n = aX_{n-1} \quad mod \quad m$$

with multiplier a and modulus m [1]. This generator was first introduced by Lehmer [8]. Up to version 1.7 of the CNCL an LCG with multiplier 65,539=2^16+3 and modulus 2^31=2,147,483,647 was provided. This generator is known as RANDU. It does not have the full possible period of 2^31-2 and it has been shown to be flawed in many respects. Also it does not have good randomness properties [1]. Furthermore the implementation in the CNCL was not correct. The formula was implemented straight forward with unsigned longs, neglecting that the product of two unsigned longs may lead to an overflow. In version 1.8 of the CNCL the RANDU was replaced by a linear congruential generator with a = 16807 and m = 2^31-1 = 2,147,483,647.

This generator provides the full possible period and was highly recommended by Park and Miller [2]. They called it a minimal standard. But be careful, also this generator is not perfect, but it may be used when performance is more important than perfect randomness. The possibility of overflow was taken into account. The algorithm of Schrage [3] was used, which allows the correct implementation of the formula on 32bit-machines. Due to the recommendation of Lewis and Orav [4] there are another five values based on the work of Fishmann and Moore [5] for the multiplier a: 950,706,376; 742,938,285; 1,226,874,159; 62,089,911 and 1,343,714,438. This values were not used in the CNCL, because it is not possible to apply the algorithm of Schrage [3] to them to avoid overflow. Their implementation needs the use of double precision floating points which impairs performance.

Seed selection

Zero is not suitable as a seed for the LCG. So if the user persists to select zero, his choice is mapped to a more useful seed. According to Knuth[1], the seed and the modulus m of an LCG should be relative prime.

Constructors:

`CNLCG(unsigned long seed = 929);`

`CNLCG(CNParam *param);`
: Initializes `CNLCG` with initial `seed`.

In addition to the member functions required by CNCL, `CNLCG` provides:

`virtual unsigned long as_long32();`
: Draws a random number. The result is an unsigned integer in the range 0 ... 2^31-1.

`virtual bool has_long32();`
: Returns FALSE because LCG produces only 31bit integer values.

`virtual void reset();`
: Resets the CNLCG to its initial state.

`void seed(unsigned long);`
: Sets the CNLCG seed value.

3.6 CNMLCG — Multiple Linear Congruence RNG

SYNOPSIS

`#include <CNCL/MLCG.h>`

TYPE

`CN_CNMLCG`

BASE CLASSES

CNRNG

DERIVED CLASSES

None

RELATED CLASSES

CNRandom

DESCRIPTION

`CNMLCG` is a multiple linear congruence random number generator class combining the results of two different CNLCGs. It is the double MLCG described by L'Ecuyer [6]. The implementation was taken from the GNU-library libg++. This generator has a fairly long period, and has been statistically analyzed to show that it gives good intersample-independence. The 'CNMLCG::CNMLCG' constructor has two parameters, both of which are seeds for the generator. Both seeds are modified to give a "better" distribution of seed digits. Thus, you can safely use values such as '0' and '1' for the seeds. Such values are mapped to values with a sufficient number and distribution of bits. Hence the constructor contains a table with 32 tested seed values.

Constructors:

```
CNMLCG();
CNMLCG(long seed1, long seed2);
CNMLCG(CNParam *param);
```

Initializes `CNMLCG` with two seeds, `seed1` and `seed2`. The default constructor sets *both* seeds to 0.

In addition to the member functions required by CNCL, `CNMLCG` provides:

`virtual unsigned long as_long32();`
: Draws a random number. The result is an unsigned integer in the range 0 ... 2^31-1.

`virtual bool has_long32();`
: Returns FALSE because CNMLCG produces only 31bit intger values.

`virtual void reset();`
: Resets the CNMLCG to its initial state.

`void seed(unsigned long s);`
: Sets the two seed values, the first one to `s` and the second one to `s + 2147483561`.

`void seed_internal(unsigned long, unsigned long);`
: Sets both seed values.

3.7 CNTausG — Tausworth RNG

SYNOPSIS

```
#include <CNCL/TausG.h>
```

TYPE

`CN_TAUSG`

BASE CLASSES

CNRNG

DERIVED CLASSES

None

RELATED CLASSES

CNRandom

DESCRIPTION

`CNTausG` is a Tausworth random number generator class. This generator is based on a paper by Tausworthe [12]. The generator, which is related to cryptographic methods, operates directly with bits to form random numbers. For further reading see [7] and [13]. A version of the Tausworthe generator was also studied by Richter [11]. The implementation of the Tausworthe Generator of the CNCL works with a fixed seed. Nevertheless the constructor accepts a seed. The effect is, that the first random numbers are thrown away as long as the quantity of numbers drawn is less than the seed. The main advantage of this generator is that it can easily be implemented as a fast hardware generator. Statistical tests [11] have shown some flaws of this generator so that its use is not recommended.

Constructors:

```
CNTausG();
CNTausG(CNParam *param);
```

Initializes `CNTausG`.

In addition to the member functions required by CNCL, `CNTausG` provides:

`virtual unsigned long as_long32();`
: Draws a random number. The result is an unsigned integer in the range 0 ... 2^32-1.

`virtual bool has_long32();`
: Returns TRUE because CNTausG is able to produce 32bit integer values.

`virtual void reset();`
: Resets the CNTausG to its initial state.

3.8 CNRndInt — Random Integers

SYNOPSIS

```
#include <CNCL/RndInt.h>
```

TYPE

`CN_RNDINT`

BASE CLASSES

CNObject

DERIVED CLASSES

None

RELATED CLASSES

CNRNG

DESCRIPTION

`CNRndInt` generates uniform distributed random integers in a given interval. The result is the same as provided by the `CNDiscUniform` distribution, but `CNRndInt` is more efficient.

BEWARE: do NOT use the `CNLCG` RNG as a base generator for `CNRndInt`.

Constructors:

```
CNRndInt();
CNRndInt(CNParam *param);
CNRndInt(long low, long high, CNRNG *gen);
CNRndInt(long high, CNRNG *gen);
CNRndInt(CNRNG *gen);
```

Initializes a `CNRndInt` with base RNG `gen` and upper/lower interval limits `low`/`high`.

In addition to the member functions required by CNCL, `CNRndInt` provides:

`CNRNG *generator() const;`

`CNRNG *generator(CNRNG *gen);`
: Gets/sets the base CNRNG used by `CNRndInt`.

`long low() const;`

`long high() const;`

`long low(long x);`

`long high(long x);`
: Gets/sets the upper/lower interval limits.

`long operator()();`

`long operator()(long high);`

`long operator()(long low, long high);`

`long as_long();`

`long as_long(long high);`

`long as_long(long low, long high);`
: Draws a `long` random integer. Interval limits may be passed as optional parameters.

`int as_int();`

`int as_int(long high);`

`int as_int(long low, long high);`
: Draws a `int` random integer. Interval limits may be passed as optional parameters.

3.9 CNRandom — Abstract Random Distribution Base Class

SYNOPSIS

`#include <CNCL/Random.h>`

TYPE

`CN_RANDOM`

BASE CLASSES

CNObject

DERIVED CLASSES

CNBeta, CNBinomial, CNDeterm, CNDiracTab, CNDiscUniform, CNErlang, CNGeometric, CNHyperExp, CNHyperGeom, CNInterTab, CNLogNormal, CNMDeterm, CNNegExp, CNNormal, CNPoisson, CNRandomMix, CNRayleigh, CNRice, CNTab, CNUniform, CNWeibull

RELATED CLASSES

CNRNG

DESCRIPTION

`CNRandom` is the abstract base class for all CNCL random number distributions. It defines a common interface to access to all derived RNG classes in a common way.

Constructors:

`CNRandom(CNRNG *gen);`
`CNRandom(CNParam *param);`
Initializes `CNRandom` with a base random number generator.

In addition to the member functions required by CNCL, `CNRandom` provides:

`CNRNG *generator();`
Returns a pointer to the actually used CNRNG.

`void generator(CNRNG *gen);`
    Sets the CNRNG used by `CNRandom` to `gen`.

`virtual double operator() () = 0;`

`double draw();`
    Draws a random number from the distribution. The operator `()` *must* be defined in the derived classes.

3.10 CNBeta — Beta Distribution

SYNOPSIS

`#include <CNCL/Beta.h>`

TYPE

`CN_BETA`

BASE CLASSES

CNRandom

DERIVED CLASSES

None

RELATED CLASSES

CNRNG

DESCRIPTION

`CNBeta` is a class for generating beta distributed random numbers.

Constructors:

`CNBeta();`

`CNBeta(CNParam *param);`

`CNBeta(long a, long b, CNRNG *gen);`
: Initializes a `CNBeta` distribution with a base random number generator `gen` and the parameters `a` and `b`.

In addition to the member functions required by CNCL, `CNBeta` provides:

`virtual double operator() ();`
: Draws a beta distributed random number.

3.11 CNBinomial — Binomial Distribution

SYNOPSIS

```
#include <CNCL/Binomial.h>
```

TYPE

`CN_BINOMIAL`

BASE CLASSES

CNRandom

DERIVED CLASSES

None

RELATED CLASSES

CNRNG

DESCRIPTION

`CNBinomial` is a class for generating binomial distributed random numbers.

Constructors:

`CNBinomial();`
`CNBinomial(CNParam *param);`
`CNBinomial(int n, double u, CNRNG *gen);`
Initializes a `CNBinomial` distribution with a base random number generator `gen`, and the parameters `n` and `u`.

In addition to the member functions required by CNCL, `CNBinomial` provides:

`int n();`
`int n(int xn);`
`double u();`

`double u(double xu);`

Gets/sets the values for n and u.

`virtual double operator() ();`

Draws a binomial distributed random number.

3.12 CNDeterm — Deterministic Distribution

SYNOPSIS

```
#include <CNCL/Determ.h>
```

TYPE

`CN_DETERM`

BASE CLASSES

CNRandom

DERIVED CLASSES

None

RELATED CLASSES

CNRNG

DESCRIPTION

`CNDeterm` is a class for generating deterministic "random" numbers, i.e. it always generates the same value. This is useful for mixed distributions.

Constructors:

```
CNDeterm();
CNDeterm(CNParam *param);
CNDeterm(double value, CNRNG *gen);
```

Initializes a `CNDeterm` distribution with `value`.

In addition to the member functions required by CNCL, `CNDeterm` provides:

```
double value();
double mean();
double value(double x);
```

`double mean(double x);`
Gets/sets the *deterministic* value.

`virtual double operator() ();`
Draws a deterministic (i.e. not random) number.

3.13 CNDiracTab — Distribution from Table of CDF

SYNOPSIS

```
#include <CNCL/DiracTab.h>
```

TYPE

`CN_DIRACTAB`

BASE CLASSES

CNRandom

DERIVED CLASSES

None

RELATED CLASSES

CNRNG

DESCRIPTION

`CNDiracTab` generates random numbers according to a table with the distribution function.

Constructors:

`CNDiracTab();`
`CNDiracTab(CNParam *param);`
`CNDiracTab(CNDiracTabEntry *tab, long length, CNRNG *gen);`

Initializes a `CNDiracTab` distribution with a base random number generator `gen` and a table `tab` with length `length`.

Example:

```
CNDiracTabEntry datafield[] = {{ 2, 0.1 },{ 4, 0.3 }, { 6, 0.6 }};
CNDiracTab example(datafield, 3, &generator);
```

In addition to the member functions required by CNCL, `CNDiracTab` provides:

`virtual double operator() ();`

Draws a random number.

3.14 CNDiscUniform — Discrete Uniform Distribution

SYNOPSIS

```
#include <CNCL/DiscUniform.h>
```

TYPE

`CN_DISCUNIFORM`

BASE CLASSES

CNRandom

DERIVED CLASSES

None

RELATED CLASSES

CNRNG

DESCRIPTION

`CNDiscUniform` is a discrete uniform distribution with interval limits `low` and `high`. It generates discrete (i.e. integer) values.

Constructors:

`CNDiscUniform();`

`CNDiscUniform(CNParam *param);`

`CNDiscUniform(long low, long high, CNRNG *gen);`
: Initializes a `CNDiscUniform` distribution with a base random number generator `gen` and the parameters `low` and `high`.

In addition to the member functions required by CNCL, `CNDiscUniform` provides:

`long low();`

`long low(long x);`

`long high();`

`long high(long x);`
    Gets/sets the values for the interval limits.

`virtual double operator() ();`
    Draws a discrete uniform distributed random number.

3.15 CNErlang — Erlang-k Distribution

SYNOPSIS

```
#include <CNCL/Erlang.h>
```

TYPE

`CN_ERLANG`

BASE CLASSES

CNRandom

DERIVED CLASSES

None

RELATED CLASSES

CNRNG

DESCRIPTION

`CNErlang` is a class for generating Erlang-k distributed random numbers.

Constructors:

`CNErlang();`
`CNErlang(CNParam *param);`
`CNErlang(double mean, double variance, CNRNG *gen);`
: Initializes a `CNErlang` distribution with a base random number generator `gen`, mean value `mean`, and variance `variance`.

In addition to the member functions required by CNCL, `CNErlang` provides:

`double mean();`
`double mean(double x);`
`double variance();`

`double variance(double x);`
Gets/sets the values for mean and variance.

`virtual double operator() ();`
Draws a Erlang-k distributed random number.

3.16 CNGeometric — Geometric Distributed Random Numbers

SYNOPSIS

```
#include <CNCL/Geometric.h>
```

TYPE

`CN_GEOMETRIC`

BASE CLASSES

CNRandom

DERIVED CLASSES

None

RELATED CLASSES

CNRNG

DESCRIPTION

`CNGeometric` is a class for generating geometric distributed random numbers.

Constructors:

`CNGeometric();`

`CNGeometric(CNParam *param);`

`CNGeometric(double mean, CNRNG *gen);`

Initializes a `CNGeometric` distribution with a base random number generator `gen` and the value `mean` as a parameter for the random generator.

NOTE: Please do NOT take this parameter as the distrbution's mean ('MEAN') value. This can be calculated as:

```
MEAN = 1 / ( 1 - mean )
```

In addition to the member functions required by CNCL, `CNGeometric` provides:

`double mean();`

`double mean(double x);`

Gets/sets the values for mean.

`virtual double operator() ();`

Draws a geometric distributed random number.

3.17 CNHyperExp — Hyperexponential Distribution

SYNOPSIS

```
#include <CNCL/HyperExp.h>
```

TYPE

`CN_HYPEREXP`

BASE CLASSES

CNRandom

DERIVED CLASSES

None

RELATED CLASSES

CNRNG

DESCRIPTION

`CNHyperExp` is a class for generating hyperexponential distributed random numbers. `CNHyperExp` is a mixed distribution of two negative exponential distributions (H2).

Constructors:

`CNHyperExp();`

`CNHyperExp(CNParam *param);`

`CNHyperExp(double p, double m1, double m2, CNRNG *gen);`

Initializes a `CNHyperExp` distribution with a base random number generator `gen`, the mixprobability `p` and the intensity parameters `m1` and `m2`.

In addition to the member functions required by CNCL, `CNHyperExp` provides:

`virtual double operator() ();`

Draws a hyperexponential distributed random number.

3.18 CNHyperGeom — Hypergeometrical Distribution

SYNOPSIS

`#include <CNCL/HyperGeom.h>`

TYPE

`CN_HYPERGEOM`

BASE CLASSES

CNRandom

DERIVED CLASSES

None

RELATED CLASSES

CNRNG

DESCRIPTION

`CNHyperGeom` is a class for generating hypergeometrical distributed random numbers.

Constructors:

`CNHyperGeom();`
`CNHyperGeom(CNParam *param);`
`CNHyperGeom(double mean, double variance, CNRNG *gen);`
: Initializes a `CNHyperGeom` distribution with a base random number generator `gen`, mean value `mean` and variance `variance`.

In addition to the member functions required by CNCL, `CNHyperGeom` provides:

`double mean();`
`double mean(double x);`
`double variance();`

`double variance(double x);`

Gets/sets the values for mean and variance.

`virtual double operator() ();`

Draws a hypergeometrical distributed random number.

3.19 CNInterTab – Distribution from Table of CDF (Interpolated)

SYNOPSIS

`#include <CNCL/InterTab.h>`

TYPE

`CN_INTERTAB`

BASE CLASSES

CNTab

DERIVED CLASSES

None

RELATED CLASSES

CNRNG

DESCRIPTION

`CNInterTab` is a class for generating random numbers from a table. It works as `CNTab`, but the values are interpolated between the table entries.

Constructors:

`CNInterTab();`

`CNInterTab(CNParam *param);`

`CNInterTab(double *tab, long length, CNRNG *gen);`

Initializes a `CNInterTab` distribution with a base random number generator `gen`, a table with the samples `tab` and the length of the table `length`.

Example:

```
double datafield[]={ 2, 4, 6, 8, 10, 12 };
CNInterTab ex(datafield, 6, generator);
```

In addition to the member functions required by CNCL, `CNInterTab` provides:

`virtual double operator() ();`

Draws a random number.

3.20 CNLogNormal — Log-normal Distribution

SYNOPSIS

```
#include <CNCL/LogNormal.h>
```

TYPE

`CN_LOGNORMAL`

BASE CLASSES

CNNormal

DERIVED CLASSES

None

RELATED CLASSES

CNRNG

DESCRIPTION

`CNLogNormal` is a class for generating logarithmic normal distributed random numbers.

Constructors:

```
CNLogNormal();
CNLogNormal(CNParam *param);
CNLogNormal(double mean, double variance, CNRNG *gen);
```

Initializes a `CNLogNormal` distribution with a base random number generator `gen`, mean value `mean` and variance `variance`.

In addition to the member functions required by CNCL, `CNLogNormal` provides:

```
double mean();
double mean(double x);
double variance();
```

`double variance(double x);`

Gets/sets the values for mean and variance.

`virtual double operator() ();`

Draws a logarithmic normal distributed random number.

3.21 CNMDeterm — Random Mix of Deterministic Values

SYNOPSIS

`#include <CNCL/MDeterm.h>`

TYPE

`CN_MDETERM`

BASE CLASSES

CNRandom

DERIVED CLASSES

None

RELATED CLASSES

CNRNG

DESCRIPTION

`CNMDeterm` is a class for generating a random mix of two deterministic values.

Constructors:

`CNMDeterm();`
`CNMDeterm(CNParam *param);`
`CNMDeterm(double m, double d1, double d2, CNRNG *gen);`
: Initializes a `CNMDeterm` distribution with a base random number generator `gen`, a mixing parameter `m` and two values `d1` and `d2`.

In addition to the member functions required by CNCL, `CNMDeterm` provides:

`virtual double operator() ();`
: Draws a random number.

3.22 CNNegExp — Negative Exponential Distribution

SYNOPSIS

```
#include <CNCL/NegExp.h>
```

TYPE

`CN_NEGEXP`

BASE CLASSES

CNRandom

DERIVED CLASSES

None

RELATED CLASSES

CNRNG

DESCRIPTION

`CNNegExp` is a class for generating negative exponential distributed random numbers.

Constructors:

`CNNegExp();`
`CNNegExp(CNParam *param);`
`CNNegExp(double mean, CNRNG *gen);`
Initializes a `CNNegExp` distribution with a base random number generator `gen` and mean value `mean`.

In addition to the member functions required by CNCL, `CNNegExp` provides:

`double mean();`
`double mean(double x);`
Gets/sets the values for mean.

`virtual double operator() ();`
Draws a negative exponential distributed random number.

3.23 CNNormal — Normal Distribution

SYNOPSIS

`#include <CNCL/Normal.h>`

TYPE

`CN_NORMAL`

BASE CLASSES

CNRandom

DERIVED CLASSES

CNRayleigh, CNRice

RELATED CLASSES

CNRNG

DESCRIPTION

`CNNormal` is a class for generating normal (Gaussian) distributed random numbers.

Constructors:

`CNNormal();`

`CNNormal(CNParam *param);`

`CNNormal(double mean, double variance, CNRNG *gen);`

Initializes a `CNNormal` distribution with a base random number generator `gen`, mean value `mean` and variance `variance`.

In addition to the member functions required by CNCL, `CNNormal` provides:

`double mean();`

`double mean(double x);`

`double variance();`

`double variance(double x);`
Gets/sets the values for mean and variance.

`virtual double operator() ();`
Draws a normal distributed random number.

3.24 CNPoisson — Poisson Distribution

SYNOPSIS

`#include <CNCL/Poisson.h>`

TYPE

`CN_POISSON`

BASE CLASSES

CNRandom

DERIVED CLASSES

None

RELATED CLASSES

CNRNG

DESCRIPTION

`CNPoisson` is a class for generating Poisson distributed random numbers.

Constructors:

`CNPoisson();`
`CNPoisson(CNParam *param);`
`CNPoisson(double mean, CNRNG *gen);`
: Initializes a `CNPoisson` distribution with a base random number generator `gen` and mean value `mean`.

In addition to the member functions required by CNCL, `CNPoisson` provides:

`double mean();`
`double mean(double x);`
: Gets/sets the values for mean.

`virtual double operator() ();`
: Draws a Poisson distributed random number.

3.25 CNRandomMix — Mix of Several `CNRandom` Distributions

SYNOPSIS

```
#include <CNCL/RandomMix.h>
```

TYPE

`CN_RANDOMMIX`

BASE CLASSES

CNRandom

DERIVED CLASSES

None

RELATED CLASSES

CNRNG

DESCRIPTION

`CNRandomMix` generates random numbers from a mix of several `CNRandom` distributions.

Constructors:

`CNRandomMix();`

`CNRandomMix(CNParam *param);`

`CNRandomMix(CNRandomMixEntry *tab, int length, CNRNG *gen);`

Initializes a `CNRandomMix` distribution with a base random number generator `gen` and a table of several distributions.

The following example shows how to use a `CNRandomMixEntry` table with 3 different distributions and their corresponding probabilities for a `CNRandomMix` distribution:

```
CNLogNormal ex1(10, 0.5, &generator1);
CNNormal    ex2(12, 0.3, &generator2);
CNHyperExp  ex3(10, 0.5, &generator3);

CNRandomMixEntry tab[] = {
```

```
    { 0.1, &ex1 }, { 0.5, &ex2 }, { 0.4 &ex3 }
};

CNRandomMix ex(tab, 3, &generator);
```

In addition to the member functions required by CNCL, `CNRandomMix` provides:

`virtual double operator() ();`
: Draws a random number.

3.26 CNRayleigh — Rayleigh Distribution

SYNOPSIS

`#include <CNCL/Rayleigh.h>`

TYPE

`CN_RAYLEIGH`

BASE CLASSES

CNNormal

DERIVED CLASSES

None

RELATED CLASSES

CNRNG

DESCRIPTION

`CNRayleigh` is a class for generating Rayleigh distributed random numbers.

Constructors:

```
CNRayleigh();
CNRayleigh(CNParam *param);
CNRayleigh(double variance, CNRNG *gen);
```

Initializes a `CNRayleigh` distribution with a base random number generator `gen` and variance `variance`.

The variance is the one passed to the `CNNormal` base class, not the actual value of the Rayleigh distribution.

In addition to the member functions required by CNCL, `CNRayleigh` provides:

```
virtual double operator() ();
```

Draws a Rayleigh distributed random number.

3.27 CNRice — Rice Distribution

SYNOPSIS

```
#include <CNCL/Rice.h>
```

TYPE

`CN_RICE`

BASE CLASSES

CNNormal

DERIVED CLASSES

None

RELATED CLASSES

CNRNG

DESCRIPTION

`CNRice` is a class for generating Rice distributed random numbers.

Constructors:

`CNRice();`
`CNRice(CNParam *param);`
`CNRice(double mean, double variance, CNRNG *gen);`

Initializes a `CNRice` distribution with a base random number generator `gen`, mean value `mean` and variance `variance`.

The mean value and variance are those passed to the `CNNormal` base class, not the actual values of the Rice distribution.

In addition to the member functions required by CNCL, `CNRice` provides:

`virtual double operator() ();`

Draws a Rice distributed random number.

3.28 CNTab — Distribution from Table of CDF

SYNOPSIS

```
#include <CNCL/Tab.h>
```

TYPE

CN_TAB

BASE CLASSES

CNRandom

DERIVED CLASSES

CNInterTab

RELATED CLASSES

CNRNG

DESCRIPTION

`CNTab` requires a table with samples.

Constructors:

`CNTab();`

`CNTab(CNParam *param);`

`CNTab(double *tab, long length, CNRNG *gen);`
Initializes a `CNTab` distribution with a base random number generator `gen`, a table of values `tab`, and length `length`.
Example:

```
double datafield[]={ 2, 4, 6, 8, 10, 12 };
CNTab ex(datafield, 6, &generator);
```

In addition to the member functions required by CNCL, `CNTab` provides:

`virtual double operator() ();`
Draws a random number.

3.29 CNUniform — Uniform Distribution

SYNOPSIS

`#include <CNCL/Uniform.h>`

TYPE

`CN_UNIFORM`

BASE CLASSES

CNRandom

DERIVED CLASSES

None

RELATED CLASSES

CNRNG

DESCRIPTION

`CNUniform` generates uniform distributed random numbers within the interval limits `low` and `high`.

Constructors:

`CNUniform();`
`CNUniform(CNParam *param);`
`CNUniform(double low, double high, CNRNG *gen);`
  Initializes a `CNUniform` distribution with a base random number generator `gen` and interval limits `low` and `high`.

In addition to the member functions required by CNCL, `CNUniform` provides:

`double low();`
`double low(double x);`

`double high();`

`double high(double x);`

Gets/sets the values for the limits.

`virtual double operator() ();`

Draws a uniform distributed random number.

3.30 CNWeibull — Weibull Distribution

SYNOPSIS

```
#include <CNCL/Weibull.h>
```

TYPE

`CN_WEIBULL`

BASE CLASSES

CNRandom

DERIVED CLASSES

None

RELATED CLASSES

CNRNG

DESCRIPTION

`CNWeibull` generates Weibull distributed random numbers with a form factor `alpha` and a scale factor `beta`.

Constructors:

`CNWeibull();`

`CNWeibull(CNParam *param);`

`CNWeibull(double alpha, double beta, CNRNG *gen);`
: Initializes a `CNWeibull` distribution with a base random number generator `gen` and parameters `alpha` and `beta`.

In addition to the member functions required by CNCL, `CNWeibull` provides:

`double alpha();`

`double alpha(double x);`

`double beta();`

`double beta(double x);`

Gets/sets the values for alpha and beta.

`virtual double operator() ();`

Draws a Weibull distributed random number.

4 Statistical Evaluation in CNCL

CNCL provides essentially six methods for a statistical evaluation of simulation results:

- Moments: mean, variance, coefficient of variation etc., without error measure (`CNMoments`)
- Moments with Time Weights: mean, variance, coefficient of variation etc., without error measure (`CNMomentsTime`)
- Confidence: mean, variance, coefficient of variation etc., with confidence interval measure measure (`CNConfidence`)
- Limited-Relative-Error (LRE): distribution function, complementary distribution function, and local correlation coefficient with error measure (`CNLREF`, `CNLREG`)
- Discrete-Limited-Relative-Error (DLRE): same as LRE, but only for discrete (complementary) distribution functions with well–known x–values
- Batch-Means: distribution function with relative Bayes error, histogram, confidence interval; mean (with relative Bayes error) and variance of the group means (`CNBatchMeans`).
- Histogram: reduction of statistical data to a simple histogram.

All classes are derived from the base class `CNStatistics`.

4.1 CNStatistics — Abstract Statistics Base Class

SYNOPSIS

```
#include <CNCL/Statistics.h>
```

TYPE

`CN_STATISTICS`

BASE CLASSES

CNObject

DERIVED CLASSES

CNMoments, CNLRE, CNDLRE, CNBatches

RELATED CLASSES

None

DESCRIPTION

`CNStatistics` is the base class of all statistics classes. It defines the common interface.

Constructors:

`CNStatistics(const char* aName = NIL, const char* aDescription = NIL);`
`CNStatistics(CNParam *param);`
: Initializes `CNStatistics`. Optionally, you can specify `aName` and `aDescription` of the statistical evaluator.

The evaluation phases and types supported by `CNStatistics` are:

`enum Phase { INITIALIZE=0, ITERATE=1, END=2 };`
: Different phases of a statistical evaluation with their settings.

`enum Type { DF=0, CDF=1, PF=2 };`
: `DF` as distribution function, `CDF` as complementary distribution function and `PF` as probability function.

In addition to the member functions required by CNCL, `CNStatistics` provides:

`virtual void put( double ) = 0;`
: Input of a value for statistical evaluation.

`virtual double mean() const = 0;`
: Returns the mean value of the input sequence.

`virtual double variance() const = 0;`
: Returns the variance of the input sequence.

`virtual long trials() const = 0;`
: Returns the number of evaluated values.

`virtual double min() const = 0;`
: Returns the minimum of all evaluated values.

`virtual double max() const = 0;`
: Returns the maximum of all evaluated values.

`virtual bool end() const = 0;`
: Returns `TRUE` if end of evaluation is reached else `FALSE`.
NOTE: In case of a `CNMoments` evaluation, the return value is always `FALSE`.

`virtual void reset() = 0;`
: Resets the evaluation.

`virtual Phase status() const = 0;`
: Returns the state of evaluation;

4.2 CNMoments — Moments Evaluation

SYNOPSIS

```
#include <CNCL/Moments.h>
```

TYPE

`CN_MOMENTS`

BASE CLASSES

CNStatistics

DERIVED CLASSES

None

RELATED CLASSES

CNMomentsTime CNConfidence

DESCRIPTION

The `CNMoments` class yields the moments of an input sequence:

- mean,
- variance and relative variance (squared coefficient of variation),
- 2nd and 3rd zero moment,
- 3rd central moment,
- deviation and relative deviation (coefficient of variation),
- skewness.

Constructors:

`CNMoments(CNParam *param)`
`CNMoments(const char* aName = NIL, const char* aDescription = NIL);`

Initializes a `CNMoments` evaluation.
Optionally, you can specify `aName` and `aDescription` of the statistical evaluator.
with `new_name` as name of evaluation.

In addition to the member functions required by CNCL and `CNStatistics`, `CNMoments` provides:

`virtual void put( double x_i, double w_i);`
: Input of a weighted value `x_i` for statistical evaluation. If no weight `w_i` is specified, `1.0` is used as a default value.

`virtual double mean() const;`
: Returns mean of the input values.

`double variance() const;`
: Returns variance.

`double M_2() const;`
: Returns 2nd moment.

`double M_3() const;`
: Returns 3rd moment.

`double Z_3() const;`
: Returns 3rd central moment.

`double skewness() const;`
: Returns skewness.

`double relative_variance() const;`
: Returns relative variance (squared coefficient of variation).

`double relative_deviation() const;`
: Returns relative deviation (coefficient of variation).

4.3 CNMomentsTime — Moments Evaluation with Time Weights

SYNOPSIS

```
#include <CNCL/MomentsTime.h>
```

TYPE

`CN_MOMENTSTIME`

BASE CLASSES

CNStatistics

DERIVED CLASSES

None

RELATED CLASSES

CNMoments

DESCRIPTION

The `CNMomentsTime` class yields the moments of an time-weighted input sequence:

- mean,
- variance and relative variance (squared coefficient of variation),
- 2nd and 3rd zero moment,
- 3rd central moment,
- deviation and relative deviation (coefficient of variation),
- skewness.

When specifying an input value, you also have to specify the current time. The time span from the last input to this input is used as the input value's weight.

Constructors:

`CNMomentsTime(CNParam *param)`

`CNMomentsTime(const char* aName = NIL, const char* aDescription = NIL);`
: Initializes a `CNMomentsTime` evaluation. Optionally, you can specify `aName` and `aDescription` of the statistical evaluator.

In addition to the member functions required by CNCL and `CNStatistics`, `CNMomentsTime` provides:

`virtual void put( double x_i, CNSimTime put_time);`
: Input of a weighted value `x_i` for statistical evaluation. You also have to specify the current time. The time span from the last input to this input is used as the input value's weight.

`virtual double mean() const;`
: Returns mean of the input values.

`double variance() const;`
: Returns variance.

`double M_2() const;`
: Returns 2nd moment.

`double M_3() const;`
: Returns 3rd moment.

`double Z_3() const;`
: Returns 3rd central moment.

`double skewness() const;`
: Returns skewness.

`double relative_variance() const;`
: Returns relative variance (squared coefficient of variation).

`double relative_deviation() const;`
: Returns relative deviation (coefficient of variation).

4.4 CNConfidence — Evaluation with Confidence Intervals

SYNOPSIS

`#include <CNCL/Confidence.h>`

TYPE

`CN_CONFIDENCE`

BASE CLASSES

CNStatistics

DERIVED CLASSES

None

RELATED CLASSES

CNMoments CNMomentsTime

DESCRIPTION

The `CNConfidence` class yields the moments of an input sequence:

- mean,
- variance and relative variance (squared coefficient of variation),
- 2nd and 3rd zero moment,
- 3rd central moment,
- deviation and relative deviation (coefficient of variation),
- skewness.

Constructors:

`CNConfidence(CNParam *param)`

`CNConfidence(const char* aName = NIL, const char* aDescription = NIL);`

Initializes a `CNConfidence` evaluation. Optionally, you can specify `aName` and `aDescription` of the statistical evaluator.

In addition to the member functions required by CNCL and `CNStatistics`, `CNConfidence` provides:

`virtual double mean() const;`
: Returns mean of the input values.

`double variance() const;`
: Returns variance.

`double M_2() const;`
: Returns 2nd moment.

`double M_3() const;`
: Returns 3rd moment.

`double Z_3() const;`
: Returns 3rd central moment.

`double skewness() const;`
: Returns skewness.

`double relative_variance() const;`
: Returns relative variance (squared coefficient of variation).

`double relative_deviation() const;`
: Returns relative deviation (coefficient of variation).

`double z_level(double conf) const;`
: Returns Z-level (argument) of error function for given confidence level (inverse error function).

`double err_level(double z_level) const;`
: Returns value of error function to given (Z-level) argument.

`double conf_level(double z_level) const;`
: Returns value of confidence probability to given (Z-level) argument.

`double low_conf_bound(double conf) const;`
: Returns lower confidence boundary to given confidence level. The estimator is determined by `mean()`.

`double hi_conf_bound(double conf) const;`
: Returns upper confidence boundary to given confidence level. The estimator is determined by `mean()`.

In general the confidence level `conf = 0.95` is quite useful. It means that in 1 out of 20 measurements the "true" value is within the confidence interval determined by `hi_conf_bound()` and `low_conf_bound()`.

4.5 CNLREF, CNLREG — Evaluation by LRE

SYNOPSIS

`#include <CNCL/LREF.h>` Distribution Function F(x)

`#include <CNCL/LREG.h>` Complementary Distribution Function G(x)

TYPE

```
CN_LREF
CN_LREG
```

BASE CLASSES

CNStatistics

DERIVED CLASSES

None

RELATED CLASSES

None

DESCRIPTION

The `CNLREF` and `CNLREG` classes provide the statistical evaluation of random sequences by the LRE algorithm. Results are the distribution (d.f.) and the complementary distribution function (c.d.f.) respectively, the local correlation coefficient, and the error of each discovered point. The simulation run time is controlled by a predefined maximum error regarding to the local correlation of the input values. For further information refer to "Effective Control of Simulation Runs by a New Algorithm for Correlated Random Sequences" by F. Schreiber, AEUe Vol. 42, pp. 347-354, 1988.

Constructors:

```
CNLREF( CNParam * param );
CNLREF( double MIN=0.01, double MAX=0.99, double MAX_ERR=0.05, int LEVEL=100,
Scale SCALE=CNLRE::LIN, int MAXSORT= 0, const char* NAME = NIL, const char* TEXT =
NIL);
```

Initializes a `CNLREF` evaluation.
Parameters:

`MIN, MAX` limits of d.f. and c.d.f respectively

`MAX_ERR` maximum error of d.f. and d.f respectively

`LEVEL` number of levels

`SCALE` scale of ordinate (`CNLRE::LIN` or `CNLRE::LOG`)

`MAXSORT` maximum size of an internal sort array

`NAME` allows to name the evaluation.

`TEXT` a short explanation of the evaluation.

`CNLREG( CNParam * param )`

`CNLREG( double MIN=0.01, double MAX=0.99, double MAX_ERR=0.05, int LEVEL=100, Scale SCALE=LIN, int MAXSORT= 0, const char* NAME = NIL, const char* TEXT = NIL)`
Initializes a `CNLREG` evaluation. The parameters are the same as described above.

In addition to the member functions required by CNCL and `CNStatistics`, `CNLREF` and `CNLREG` provide:

`void set_base(double b);`
Set base for conditional probablity.

`void change_error(double ne);`
Change desired error during simulation.

`long min_index() const;`

`long max_index() const;`
Return start and end of result arrary. Should be used together with `get_result()` to acquire online evaluation during simulation.

`const CNLRE::resultline *get_result(long lev);`
Can be used to acquire online (preliminary) results of the LRE[FG] during simulation. The parameter `lev` must be in the range `min_index` to `max_index`. A result line is a structs with the members `x`, `vf` — holds F- or G-value —, `rho`, `sigrho`, `d` — the relative error — and `nx` — the number of exact hits of `x`.

`double cur_x_lev() const;`
Returns the x-level currently calculated.

`double cur_f_lev();`

`double cur_g_lev();`
These return the current F- resp. G-level in calculation.

4.6 CNDLRE — Evaluation by Discrete–LRE

SYNOPSIS

#include <CNCL/DLREF.h> Distribution Function F(x)

#include <CNCL/DLREG.h> Complementary Distr. Function G(x)

#include <CNCL/DLREP.h> Probability Function P(x)

TYPE

CN_DLREF
CN_DLREG
CN_DLREP

BASE CLASSES

CNStatistics, CNDLRE

DERIVED CLASSES

None

RELATED CLASSES

LREF, LREG

DESCRIPTION

The Discrete Limited Relative Error algorithm (DLRE or LRE III) is a special LRE for evaluating discrete random variables. The possible values must be known before the evalution is initialized, because F(x) resp. G(x) or P(x) of each value will be estimated. So, the DLRE is not a better histogramm with an error measure. The implementation is a generalized version of the algorithm.

If a test value does not agree with any x–interval, the following warning is printed: `Warning: Wrong x value in DLRE[FGP]::put !`. Since CNCL 1.8, values are rounded in order to find a suitable interval. Thus the above error message should not occur any longer.

Constructors:

`CNDLREF( CNParam * param );`

`CNDLREF( double XMIN, double XMAX, double INT_SIZE, double MAX_ERR, double PRE_FIRST = 0.0, const char* NAME = NIL, const char* TEXT = NIL, double Fmin = 0.0, unsigned long MAX_NRV = ULONG_MAX);`

`CNDLREF( double * XVALUES, long LEVEL, double MAX_ERR, double PRE_FIRST = 0.0, const char* NAME = NIL, const char* TEXT = NIL, double Fmin = 0.0, unsigned long MAX_NRV = ULONG_MAX);`

Initializes a `CNDLREF` evaluation. There are two types of the initializing: The first constructor (second of the list above) should be used for equidistant x–values and the second for non–equidistant x–values.

Parameters:

`XMIN, XMAX`
: Minimal and maximal x–values, whose F(x) resp. G(x) shall be estimated.

`INT_SIZE`
: Size of an interval of the x–axis

`XVALUES`
: Pointer to an array of doubles, which includes the x–values.

`LEVEL`
: The number of array elements.

`MAX_ERR`
: maximum error of d.f. and d.f respectively

`PRE_FIRST`
: Predecessor of the first test value. That is necessary because of the correlation; you should not think about it too long, since a false value causes only a very small error in the measurement.

`NAME`
: allows to name the evaluation.

`TEXT`
: a short explanation of the evaluation.

`Fmin`
: minimum F-level to stop the evaluation even when the proposed error criteria aren't fulfilled. (Default is 0.0).

`MAX_NRV`
: The maximum number of test values allowed.

`CNDLREG( CNParam * param )`

`CNDLREG( double XMIN, double XMAX, double INT_SIZE, double MAX_ERR, double PRE_FIRST = 0.0, const char* NAME = NIL, const char* TEXT = NIL, double Gmin = 0.0, unsigned long MAX_NRV = ULONG_MAX);`

`CNDLREG( double * XVALUES, long LEVEL, double MAX_ERR, double PRE_FIRST = 0.0, const char* NAME = NIL, const char* TEXT = NIL, double Gmin = 0.0, unsigned long MAX_NRV = ULONG_MAX);`

Initializes a `CNDLREG` evaluation. The parameters are as described above (except Gmin which replaces Fmin, of course).

`CNDLREP( CNParam * param )`

`CNDLREP( double XMIN, double XMAX, double INT_SIZE, double MAX_ERR, double PRE_FIRST = 0.0, const char* NAME = NIL, const char* TEXT = NIL, bool force_rminusa_ok = false, unsigned long MAX_NRV = ULONG_MAX);`

`CNDLREP( double * XVALUES, long LEVEL, double MAX_ERR, double PRE_FIRST = 0.0, const char* NAME = NIL, const char* TEXT = NIL, bool force_rminusa_ok = false, unsigned long MAX_NRV = ULONG_MAX);`

Initializes a `CNDLREP` evaluation. The parameters are as described above. The new parameter `force_rminusa_ok` is used to force (hence the name) the large sample condition r-a >= 10. This condition cannot necessary be fulfilled by all kind of simulations, but most often it only results in a longer run time. To ensure the correctness of the results this option should be set TRUE whenever possible (default is FALSE).

In addition to the member functions required by CNCL and `CNStatistics`, `CNDLREF`, `CNDLREG` and `CNDLREP` provide:

`void set_base( double ba );`

Used as factor for conditional probability.

`virtual void change_error( double err );`

Change required relative error during simulation.

`virtual double cur_x_lev();`

Returns the first/last x-value whose relative error is higher than the required one.

`double cur_f_lev();`

`double cur_g_lev();`

Indicates the currently evaluated F- resp. G-Level. This may show how far the evaluation has progressed.

`virtual long min_index();`

`virtual long max_index();`

Return the lowest/highest index of the result array. These functions can be used in conjunction with the function `get_result()`.

`virtual const struct CNDLRE::resultline *get_result( long index );`

Returns a single result line. The available fields are `x`, `vf` — holds F, G or P —, `rho` — local correlation —, `sigrho`, `d` — the relative error — and `nx` — the absolut hits of x.

`virtual double f( double xt );`

`virtual double g( double xt );`

`virtual double p( double xt );`

These functions return the actual calculated F-, G-, or P-Level for the supplied x-value.

4.7 CNBatchMeans — Evaluation by Batch Means

SYNOPSIS

```
#include <CNCL/BatchMeans.h>
```

TYPE

`CN_BatchMeans`

BASE CLASSES

CNStatistics

DERIVED CLASSES

None

RELATED CLASSES

None

DESCRIPTION

The class `CNBatchMeans` provides the statistical evaluation of random sequences by the Batch-Means method. Results are the distribution (d.f.) and complementary distribution function (c.d.f.) respectively, a histogram of relative frequencies, Bayes-error and confidence intervals of each point of the d.f and the c.d.f. respectively, and of the estimated mean and the variance of the group means.

The evaluation needs at least n * h values; n is the number of batches and h the size of the batches if the constructor for fixed length evaluation is used. The evaluation is controlled by the relative Bayes error of the estimated mean when the constructor for variable length evaluation is used. For further information refer to "Principles of Discrete Event Simulation" by G.S. Fishman, J. Wiley & Sons, New York, 1978 and to "Improved Simulation by Application of the Objetive Bayes-Statistics" by F. Schreiber, AEUE, Vol. 34, pp. 234-249, 1980.

Constructors:

```
CNBatchMeans();
CNBatchMeans( CNParam *param );
```

`CNBatchMeans( double bottom, double top, long intervals,`
`long size_of_groups, long no_of_groups, short conf = 95,`
`const char* name = NIL, const char* text = NIL );`

`CNBatchMeans( double bottom, double top, long intervals,`
`long size_of_groups, double max_rel_err, short conf = 95,`
`const char* name = NIL, const char* text = NIL );`

Initializes a `CNBatchMeans` evaluation.The Parameters are:

`bottom`
`top` lower resp. upper limit of evaluated values; values beyond these limits are only counted;

`Mmax_rel_err`
defined maximum error for variable length evaluation

`no_of_groups`
number of groups (batches) for fixed length evaluation

`size_of_groups`
size of one group (batch)

`intervals`
number of intervals to use. The higher the interval number the finer is the resolution of the distribution function, but the bigger are the confidence intervals.

`name, text`
a name and a descriptive text to use for the evaluation

In addition to the member functions required by CNCL and `CNStatistics`, `CNBatchMeans` provides:

`double bayes_err() const;`
Returns the relative Bayes error of the mean.

`double sigma() const;`
Returns the deviation of the group means; can also be used as an error measure of the mean, e.g. its relative Bayes error.

`double mean_confidence() const;`
Returns the confidence interval of the mean.

`long min_index() const;`
`long max_index() const;`
Return min and max interval number (maps to one line of output of the print function).

`long groups_done() const;`
Returns the number of evaluated groups (batches). Can be used as a progress report.

`const struct CNBatchMeans::resultline *get_result(long index);`
Returns one line of the result (as output by print). The range for index is min_index <= index <= max_index. The fields of the struct resultline are x, fx — d.f., gx — c.d.f., rh — rel. probability of x, ferr, gerr — bayes error of d.f. resp. c.d.f., fconf, gconf — confidence interval.

`void change_error(double ne);`
Allows to change the maximum relative error during evaluation.

`double p(double x) const;`

`double f(double x) const;`

`double g(double x) const;`

Return probability, value of distribution function or value of complementary distribution function associated with the interval x belongs to.

`double correlation() const;`

Returns the 1st order correlation coefficient of the batch means. It should be nearly 0 in order to trust the evaluation results.

`virtual void print( Type type = CNStatistics::DF, ostream &strm = cout ) const;`

The first argument chooses between the output of d.f (`CNStatistics::DF`, default) and c.d.f. (`CNStatistics::CDF`), the second is an user defined output stream.

4.8 CNHistogram — Reduction of statistical data to an histogram.

SYNOPSIS

`#include <CNCL/Histogram.h>`

TYPE

`CN_HISTOGRAM`

BASE CLASSES

CNStatistics

DERIVED CLASSES

None

RELATED CLASSES

None

DESCRIPTION

`CNHistogram` contains functions to reduce input data to a histogram. It provides two main output formats: Either you can create an easy readable array or you can write the results of the evaluation to a file in a format that can further be processed by 'gnuplot'.

Constructors

`CNHistogram();`
Not yet implemented

`CNHistogram(CNParam *);`
Not yet implemented

```
CNHistogram(char *aName, double l_border, double u_border,
short int num, double normalization = 0.0,
char* aDescription = NIL);
```
Initializes `CNHistogram`.

The arguments are:

aName: Name of the histogram

l_border: Lower border for evaluation

u_border: Upper border for evaluation

num: Number of bins

normalization: The total number of entries in the histogram is normalized to this value. The default value '0' means, that no normalization is performed.

In addition to the member functions required by CNCL, `CNHistogram` provides:

`void put( double );`
: Input function of the histogram. The input value is sampled into the histogram.

`double mean() const;`
: Returns the mean value of the input sequence.

`double variance() const;`
: Returns the variance of the input sequence.

`long trials() const;`
: Returns the number of values sampled in the histogram.

`double min() const;`
: Returns the minimum input value.

`double max() const;`
: Returns the maximum input value.

`boolean end() const;`
: Always returns `FALSE` as an histogram evaluation never reaches an end.

`void reset();`
: Not yet implemented, but required by base class CNStatistics. (Might be implemented later if necessary).

`Phase status() const;`
: Returns the state of evaluation. Since evaluation never reaches an end, status only returns INITIALIZE (no value in histogram) or ITERATE.

`void set_norm(double);`
: Sets normalization to new value. Does not change any internal data. Only the representation of the output is changed. It is therefore possible to change normalization several times during evaluation of a histogram.

`void print_header (ostream &, char * = "") const;`
: Writes header of histogram including statistic informations into the specified output stream. All lines are prefixed with the string given in the (optional) second parameter.

`void print_histo(ostream &) const;`
: Writes the histogram data into the specified output stream in a human readable format.

`void print_file(char*);`
: Creates output file and uses print_header and print_histo to fill it.

`void plot_histo(ostream &, int = 0);`
: Writes the histogram data and the header into the specified output stream to be used as input values for gnuplot to draw the histogram. The header is written with `print_header` prefixed with the string "# " and therefore ignored by gnuplot.

The (optional) second parameter specifies the drawing style: default 0 means that all vertical lines are drawn to the value of the next step (Step-histogram) a value != 0 means that all vertical lines are drawn down to the x-axis (Box-histogram). Those output lines, that differ in both drawing styles are either beginning with '# ' (Step-style) or with a blank ' '(Box-style). Thus the drawing style can easily be converted with a text editor by changing those lines appropriately. As only points are written, the plot command in 'gnuplot' has to given with the specification `with lines`.

`void plot_file(char*, int);`
Creates output file and uses plot_histo to fill it. The second parameter specifies the drawing style used by plot_histo.

In addition the following operators are defined:

`ostream &operator << (ostream &strm, const CNHistogram &obj);`
Defines operator `<<` to provide output of CNHistogram-objects to ostream. The member functions print_header and print_histo are called for 'obj'.

`ostream &operator << (ostream &strm, const CNHistogram *obj);`
Same functionality as above.

Also defined is a static member functions, that can very easily be used to sample data from a file into a histogram

`static CNHistogram * CNHistogram::fill_histo_from_file(char * infile, double lower_bound, double upper_bound, short int number_of_bins, double normalisation, short int use_data_column);`
Data is read from file 'infile' and values from column 'use_data_column' are sampled into the histogram newly created and returned by the function. The other parameters correspond to those described above.

This function is used e.g. by the test program `tHistogram`.

5 Container Classes

CNCL's hierarchical concept and the runtime type information make it possible to provide generic containers, i.e. container that can contain any object. CNCL generic containers classes work with pointers to `CNObject`, a class with which all classes in the CNCL hierarchy are type-compatible.

5.1 CNAVLTree — AVL balanced binary search tree

SYNOPSIS

```
#include <CNCL/AVLTree.h>
```

TYPE

`CN_CNAVLTREE`

BASE CLASSES

CNObject

DERIVED CLASSES

none

RELATED CLASSES

CNAVLNode

DESCRIPTION

`CNAVLTree` is realizing a generic AVL tree. It contains nodes derived from `CNAVLNode` and organizes them in a balanced binary search tree. Such Nodes can be added to the tree, searched for and be removed.

`CNAVLTree` is an abstract base class. It provides the general algorithms needed for AVL trees, but it makes no assumptions on what kind of key the nodes will be sorted by. Usable AVL trees require derived classes that specify the key that will be used.

Constructors:

`CNAVLTree();`
`CNAVLTree(CNParam *param);`
Initializes an empty AVL tree.

Destructor:

`~CNAVLTree();`
: Deletes the tree and all `CNAVLNode` nodes still in the tree.

In addition to the member functions required by CNCL, `CNAVLTree` provides:

`bool add(CNAVLNode*);`
: Adds a node to the tree. If the key the node is sorted by is already existing in the current AVL tree, `FALSE` is returned. otherwise `TRUE` is returned.

`CNAVLNode *find();`
: Searches for a key. This is an abstract functions that may only be called by subclasses of `CNAVLTree`. The subclasses have to provide new `find()` functions that manage the key to search for. Returns a pointer to the found `CNAVLNode` on success and `NIL` on failure (key not found).

`CNAVLNode *remove();`
: Similar to `find()`, but the returned node is also removed from the tree.

`bool empty();`
: Returns `TRUE` if the tree is empty.

`void delete_all();`
: Deletes all nodes still in the tree.

`CNAVLNode *find_first();`
: Returns a pointer to the first node in the tree, i.e. the node with the lowest key. Returns `NIL` if the tree is empty.

`CNAVLNode *remove_first();`
: Similar to `find_first()`, but also removes the returned node.

`CNAVLNode *get_root();`
: Returns the root node of the tree or `NIL` is the tree is empty.

`unsigned long length() const;`
: Returns the number of nodes in the tree.

The following example shows parts of a derived class. It uses simple `long` keys to sort and find nodes. See also the example for `CNAVLNode`.

```
class IntAVLTree : public CNAVLTree
{
    friend class IntAVLNode;

  // ...

  public:  /***** Public interface **************************************/

    virtual CNAVLNode *find(long key);
    virtual CNAVLNode *remove(long key);

  private:  /***** Internal private members ******************************/

    long searchkey;
```

```
  // ...
};

// ...

CNAVLNode *IntAVLTree::find(long key) {
    searchkey = key;
    return CNAVLTree::find();
};

CNAVLNode *IntAVLTree::remove(long key) {
    searchkey = key;
    return CNAVLTree::remove();
};
```

5.2 CNAVLNode — Node for CNAVLTree

SYNOPSIS

`#include <CNCL/AVLNode.h>`

TYPE

`CN_AVLNODE`

BASE CLASSES

CNObject

DERIVED CLASSES

none

RELATED CLASSES

CNAVLTree

DESCRIPTION

`CNAVLNode` is an abstract base class for nodes of AVL trees.

Derived classes must be implemented for different key types. They must be able to compare their keys between two node and between a node and the serached key of the tree.

Constructors:

`CNAVLNode();`
`CNAVLNode(CNParam *param);`
: Initializes a new AVL node.

In addition to the member functions required by CNCL, `CNAVLNode` provides:

`virtual int compare(CNAVLNode*) = 0;`
: Compares the node's key with another node's key. Returns `-1` if the current node's key is lower, `1` of it's higher and `0` if it's equal to the other node's key.

`virtual int find(CNAVLTree*) = 0;`
: Compares the node's key with the current search key. Returns `-1` if the current node's key is lower, `1` of it's higher and `0` if it's equal to the searched key.

  Derived tree classes must provide a possibility for nodes to get the currently searched tree key, e.g. as a member variable in the tree structure.

`CNAVLNode *left();`
: Returns the root of the left sub-tree (lower keys) or `NIL` if there is no left sub-tree.

`CNAVLNode *right();`
: Returns the root of the right sub-tree (higher keys) or `NIL` if there is no right sub-tree.

The following example shows parts of a derived class. It uses simple `long` keys to sort and find nodes. See also the example for `CNAVLTree`.

```
class IntAVLNode : public CNAVLNode
{
  public:  /***** Constructors ******************************************/

    IntAVLNode(long key): key_(key) {};

  public:  /***** Public interface **************************************/

    virtual int compare(CNAVLNode*); // compare node with another one
    virtual int find(CNAVLTree*);    // compare node with searched key
    long get_key() { return key_; };

  private:  /***** Internal private members *****************************/

    long key_;

  // ...

};

// ...

int IntAVLNode::compare(CNAVLNode *n) {
    long c = IntAVLNode::cast_from_object(n)->key_;
    if (key_ < c) return -1;
    if (key_ > c) return  1;
    return 0;
};

int IntAVLNode::find(CNAVLTree *t) {
    long c = IntAVLTree::cast_from_object(t)->searchkey;
    if (key_ < c) return -1;
    if (key_ > c) return  1;
    return 0;
};
```

5.3 CNSLList — Single Linked List of Objects

SYNOPSIS

```
#include <CNCL/SLList.h>
```

TYPE

`CN_SLLIST`

BASE CLASSES

CNObject

DERIVED CLASSES

CNDLList

RELATED CLASSES

CNSLObject, CNSLIterator

DESCRIPTION

`CNSLList` is a single linked list that can contain any CNCL compatible object.

Constructors:

`CNSLList();`
`CNSLList(CNParam *param);`
: Initializes the list to empty state.

Please note, that there is NO copy constructor supplied. Any attempt to copy a CNSLList will yield a fatal error.

Destructors:

`~CNSLList();`
: Deletes the linked list and all `CNSLObject` nodes. It does NOT delete the objects referenced by the nodes.

In addition to the member functions required by CNCL, `CNSLList` provides:

`CNSLObject *first() const;`
: Returns the first node in the list or `NIL` if the list is empty.

`virtual CNSLObject *last() const;`
: Returns the last node in the list or `NIL` if the list is empty.

`CNSLObject *next(CNSLObject *link) const;`
: Returns the next node in the list, where `link` points to the current node. This may be `NIL` if the next node doesn't exist.

`virtual CNSLObject *prev(CNSLObject *link) const;`
: Returns the previous node in the list, where `link` points to the current node. This may be `NIL` if the previous node doesn't exist.

`bool empty() const;`
: Checks if `DLList` is empty.

`unsigned long length() const;`
: Returns the length (number of nodes) of this `DLList`.

`virtual CNSLObject *append(CNObject *obj);`
`virtual CNSLObject *append(CNObject &obj);`
: Adds a new node to the end of the list, referencing the object `obj`. It returns the node allocated for the object.

`virtual CNSLObject *append(CNSLObject *obj);`
: Adds an already allocated node to the list. It returns the pointer `obj`.

`virtual CNSLObject *prepend(CNObject *obj);`
`virtual CNSLObject *prepend(CNObject &obj);`
: Adds a new node to the start of the list, referencing the object `obj`. It returns the node allocated for the object.

`virtual CNSLObject *prepend(CNSLObject *obj);`
: Adds an already allocated node to the list. It returns the pointer `obj`.

`virtual CNSLObject *delete_object(CNSLObject *pos);`
: Deletes the node `pos` from the list. It returns the next node. This function deletes the node (`CNSLObject`) from the list, but it does NOT delete the object referenced by the node.

`virtual CNSLObject *remove_object(CNSLObject *pos);`
: Removes the node `pos` from the list. It returns the next node. This function does NOT delete the linked list node (`CNSLObject`) and it does NOT delete the object referenced by the node, either.

`void delete_all();`
: Deletes all nodes from the linked list and initializes the list to empty. The objects referenced by the nodes are NOT deleted.

`void delete_all_w_obj();`
: Deletes all nodes from the linked list, initializes the list to empty, AND deletes the referenced objects.

`virtual CNSLObject *insert_before(CNSLObject *pos, CNObject *obj);`
`virtual CNSLObject *insert_before(CNSLObject *pos, CNObject &obj);`
: Creates a new node for `obj` and inserts it into the list before node `pos`. It returns the new node.

`virtual CNSLObject *insert_before(CNSLObject *pos, CNSLObject *obj);`
: Inserts an already allocated node into the list before node `pos`. It returns `obj`.

`virtual CNSLObject *insert_after(CNSLObject *pos, CNObject *obj);`
`virtual CNSLObject *insert_after(CNSLObject *pos, CNObject &obj);`
: Creates a new node for `obj` and inserts it into the list after node `pos`. It returns the new node.

`virtual CNSLObject *insert_after(CNSLObject *pos, CNSLObject *obj);`
: Inserts an already allocated node into the list after node `pos`. It returns `obj`.

5.4 CNSLObject — Node of Single Linked List

SYNOPSIS

`#include <CNCL/SLObject.h>`

TYPE

`CN_SLOBJECT`

BASE CLASSES

CNObject

DERIVED CLASSES

CNDLObject

RELATED CLASSES

CNSLList, CNSLIterator

DESCRIPTION

`CNSLObject` is a node in the `CNSLList` single linked list. It contains a pointer to the next and a pointer to the referenced object.

Constructors:

```
CNSLObject();
CNSLObject(CNParam *param);
CNSLObject(CNObject *obj);
```

Initializes `CNSLObject` and optionally sets a referenced object.

`CNSLObject`s have a private destructor and can therefore only be allocated on the heap. Furthermore `CNSLObject`s cannot be copied, no copy constructor is supplied; an attempt to do so results in a runtime error.

In addition to the member functions required by CNCL, `CNSLObject` provides:

`CNSLObject *set_next(CNSLObject *p);`

`CNSLObject *next(CNSLObject *p);`

`CNSLObject *get_next();`

`CNSLObject *next();`

Sets/gets the pointer to the next node. It returns the current pointer.

`CNObject *object(CNObject *obj);`

`CNObject *object();`

`CNObject *set_object(CNObject *obj);`

`CNObject *get_object();`

Gets/sets the pointer to the referenced object. It returns the current value.

`void delete_object();`

Deletes the referenced object.

5.5 CNSLIterator — Iterator of Single Linked List

SYNOPSIS

```
#include <CNCL/SLIterator.h>
```

TYPE

`CN_SLITERATOR`

BASE CLASSES

CNObject

DERIVED CLASSES

CNDLIterator

RELATED CLASSES

CNSLList, CNSLObject

DESCRIPTION

`CNSLIterator` is an iterator to traverse a `CNSLList` single linked list.

Constructors:

`CNSLIterator();`

`CNSLIterator(CNParam *param);`
: Initializes `CNSLIterator`.

`CNSLIterator(const CNSLList *new_list);`

`CNSLIterator(const CNSLList &new_list);`
: Initializes `CNSLIterator` with linked list `list`. The iterator is reset to the first element in the list.

In addition to the member functions required by CNCL, `CNSLIterator` provides:

`void reset(const CNSLList *new_list);`

`void reset(const CNSLList &new_list);`

`void reset();`
: Resets the iterator to a new list `new_list` and/or sets the iterator to the first element in the list.

`CNObject *object()`

`CNObject *get_object()`
: Gets the referenced object from the current iterator position. It returns the object or `NIL`, if none is available.

`CNSLObject *position()`

`CNSLObject *get_position()`
: Gets the current iterator position (node in the list). It returns a pointer to the node or `NIL`, if none is available.

`void position(CNSLObject *pos)`

`void set_position(CNSLObject *pos)`
: Moves the iterator to the referenced node in the list.

`CNObject *first_object();`

`CNObject *first();`
: Sets the iterator to the first element in the list. It returns the referenced object or `NIL`, if none is available.

`CNObject *last_object();`

`CNObject *last();`
: Sets the iterator to the last element in the list. It returns the referenced object or `NIL`, if none is available.

`CNObject *next_object();`

`CNObject *next();`

`CNObject *operator ++();`

`CNObject *operator ++(int);`
: Moves the iterator to the next element in the list. It returns the current referenced object (the one BEFORE moving the iterator) or `NIL`, if none is available.

An example which shows the use of a `Iterators` object to traverse a double linked list can be found at the end of the class `CNDIterator`.

5.6 CNDLList — Doubly Linked List of Objects

SYNOPSIS

`#include <CNCL/DLList.h>`

TYPE

`CN_DLLIST`

BASE CLASSES

CNSLList

DERIVED CLASSES

None

RELATED CLASSES

CNDLObject, CNDLIterator

DESCRIPTION

`CNDLList` is a doubly linked list that can contain any CNCL compatible object.

Constructors:

`CNDLList();`
`CNDLList(CNParam *param);`
: Initializes the list to empty state.

Please note, that there is NO copy constructor supplied. Any attempt to copy a CNDLList will yield a fatal error.

Destructors:

`~CNDLList();`
: Deletes the linked list and all `CNDLObject` nodes. It does NOT delete the objects referenced by the nodes.

In addition to the member functions required by CNCL and to the functions supplied by `CNSLList`, `CNDLList` provides or defines more efficiently:

`CNDLObject *last() const;`
: Returns the last node in the list or `NIL` if the list is empty.

`CNDLObject *prev(CNDLObject *link) const;`
: Returns the previous node in the list, where `link` points to the current node. This may be `NIL` if the previous node doesn't exist.

`CNDLObject *append(CNObject *obj);`
`CNDLObject *append(CNObject &obj);`
: Adds a new node to the end of the list, referencing the object `obj`. It returns the node allocated for the object.

`CNDLObject *append(CNDLObject *obj);`
: Adds an already allocated node to the list. It returns the pointer `obj`.

`CNDLObject *insert_before(CNDLObject *pos, CNObject *obj);`
`CNDLObject *insert_before(CNDLObject *pos, CNObject &obj);`
: Creates a new node for `obj` and inserts it into the list before node `pos`. It returns the new node.

`CNDLObject *insert_before(CNDLObject *pos, CNDLObject *obj);`
: Inserts an already allocated node into the list before node `pos`. It returns `obj`.

`bool ok();`
: Checks the list for consistency. It returns `TRUE`, if the list is o.k.

5.7 CNDLObject — Node of Doubly Linked List

SYNOPSIS

`#include <CNCL/DLObject.h>`

TYPE

`CN_DLOBJECT`

BASE CLASSES

CNSLObject

DERIVED CLASSES

None

RELATED CLASSES

CNDLList, CNDLIterator

DESCRIPTION

`CNDLObject` is a node in the `CNDLList` doubly linked list. In addition to the base class it contains a pointer to the previous node.

Constructors:

```
CNDLObject();
CNDLObject(CNParam *param);
CNDLObject(CNObject *obj);
```

Initializes `CNDLObject` and optionally sets a referenced object.

`CNDLObject`s have a private destructor and can therefore only be allocated on the heap. Furthermore `CNDLObject`s cannot be copied, no copy constructor is supplied; an attempt do so results in a runtime error.

In addition to the member functions required by CNCL and to the functions declared at `CNSLObject`, `CNDLObject` provides:

`CNDLObject *prev(CNDLObject *p);`
`CNDLObject *prev();`
`CNDLObject *set_prev(CNDLObject *p);`
`CNDLObject *get_prev();`

Gets/sets the pointer to the previous node. It returns the current pointer.

5.8 CNDLIterator — Iterator of Doubly Linked List

SYNOPSIS

`#include <CNCL/DLIterator.h>`

TYPE

`CN_DLITERATOR`

BASE CLASSES

CNSLIterator

DERIVED CLASSES

None

RELATED CLASSES

CNDLList, CNDLObject

DESCRIPTION

`CNDLIterator` is an iterator to traverse a `CNDLList` doubly linked list.

Constructors:

`CNDLIterator();`
`CNDLIterator(CNParam *param);`
: Initializes `CNDLIterator`.

`CNDLIterator(const CNDLList *new_list);`
`CNDLIterator(const CNDLList &new_list);`
: Initializes `CNDLIterator` with linked list `list`. The iterator is reset to the first element in the list.

In addition to the member functions required by CNCL, `CNDLIterator` provides or defines more efficiently:

`void reset(const CNDLList *new_list);`

`void reset(const CNDLList &new_list);`

`void reset();`

Resets the iterator to a new list `new_list` and/or sets the iterator to the first element in the list.

`CNDLObject *position()`

`CNDLObject *get_position()`

Gets the current iterator position (node in the list). It returns a pointer to the node or `NIL`, if none is available.

`void position(CNDLObject *pos)`

`void set_position(CNDLObject *pos)`

Moves the iterator to the referenced node in the list.

`CNObject *last_object();`

`CNObject *last();`

Sets the iterator to the last element in the list. It returns the referenced object or `NIL`, if none is available.

`CNObject *prev_object();`

`CNObject *prev();`

`CNObject *operator --();`

`CNObject *operator --(int);`

Moves the iterator to the previous element in the list. It returns the current referenced object (the one BEFORE moving the iterator) or `NIL`, if none is available.

The following examples show how to use a `CNDLIterator` object to traverse a linked list:

Forward:

```
CNDLList list;

...

CNDLIterator trav(list);
CNObject *obj;

while(obj = trav++)
{
    // Do something with obj ...
}
```

Alternate forward:

```
for(trav.reset(list); obj=trav.object(); trav.next())
{
    // ...
}
```

Backward:

```
for(trav.last(); obj=trav.object(); trav--)
{
    // ...
}
```

5.9 CNQueue — Abstract Queue Base Class

SYNOPSIS

`#include <CNCL/Queue.h>`

TYPE

`CN_QUEUE`

BASE CLASSES

CNObject

DERIVED CLASSES

CNPrioQueueFIFO, CNQueueFIFO, CNQueueLIFO, CNQueueRandom, CNQueueSPT,

CNSink

RELATED CLASSES

CNJob, CNStack

DESCRIPTION

`CNQueue` is a queue of any CNCL compatible object.

Constructors:

`CNQueue();`
`CNQueue(CNParam *param);`
Initialize the queue.

In addition to the member functions required by CNCL, `CNQueue` provides the following abstract member function, which must be implemented by the derived classes:

`virtual bool empty() const = 0;`
Returns `TRUE`, if the queue is empty.

`virtual bool full() const = 0;`
Returns `TRUE`, if the queue is full.

`virtual int length() const = 0;`
Returns the actual queue length.

`virtual void put(CNObject *obj) = 0;`
`void put(CNObject &obj);`
Puts an object into the queue.

`virtual CNObject *get() = 0;`
Retrieves an object from the queue.

`virtual CNObject *peek() = 0;`
Retrieves an object from the queue. Unlike `get()`, the object is not removed from the queue.

`virtual void delete_all() = 0;`
Deletes all objects in the queue.

5.10 CNQueueFIFO — FIFO Queue

SYNOPSIS

`#include<CNCL/QueueFIFO.h>`

TYPE

`CN_QUEUEFIFO`

BASE CLASSES

CNQueue

DERIVED CLASSES

none

RELATED CLASSES

CNDLList, CNQueueLIFO, CNQueueRandom, CNQueueSPT, CNPrioQueueFIFO, CNSink, CNJob, CNStack

DESCIPTION

`CNQueueFIFO` is a queue, implemented as a doubly linked list, that can contain any number (well, sort of ... ;-) CNCL compatible objects. The queueing strategy is FIFO (First In, First Out).

Constructors:

`CNQueueFIFO();`

`CNQueueFIFO(CNParam *param);`
: Initialize the FIFO-queue to an empty state.

In addition to the member functions required by CNCL, `CNQueueFIFO` provides:

`virtual bool empty() const;`
: Returns `TRUE`, if the queue is empty.

`virtual bool full() const;`
: Always returns `TRUE`.

`virtual int length() const;`
Returns the actual queue length.

`virtual void put(CNObject *obj);`
Puts an object into the queue.

`virtual CNObject *get();`
Retrieves an object from the queue.

`virtual CNObject *peek();`
Retrieves an object from the queue. Unlike `get()`, the object is not removed from the queue.

`virtual void delete_all();`
Deletes all objects in the queue.

5.11 CNQueueLIFO — LIFO queue

SYNOPSIS

```
#include <CNCL/QueueLIFO.h>
```

TYPE

`CN_QUEUELIFO`

BASE CLASSES

CNQueue

DERIVED CLASSES

None

RELATED CLASSES

CNDLList, CNQueueFIFO, CNQueueRandom, CNQueueSPT, CNPrioQueueFIFO, CNSink, CNJob, CNStack

DESCRIPTION

`CNQueueLIFO` is a queue, implemented as a doubly linked list, that can contain any number (well, sort of ... ;-) CNCL compatible objects. The queueing strategy is LIFO (Last In, First Out).

Constructors:

`CNQueueLIFO();`
`CNQueueLIFO(CNParam *param);`
: Initialize the LIFO-queue to an empty state.

In addition to the member functions required by CNCL, `CNQueueLIFO` provides:

`virtual bool empty() const;`
: Returns `TRUE`, if the queue is empty.

`virtual bool full() const;`
: Always returns `FALSE`.

`virtual int length() const;`
: Returns the actual queue length.

`virtual void put(CNObject *obj);`
: Puts an object into the queue.

`virtual CNObject *get();`
: Retrieves an object from the queue.

`virtual CNObject *peek();`
: Retrieves an object from the queue. Unlike `get()`, the object is not removed from the queue.

`virtual void delete_all();`
: Deletes all objects in the queue.

5.12 CNQueueRandom — Random queue

SYNOPSIS

`#include <CNCL/QueueRandom.h>`

TYPE

`CN_QUEUERANDOM`

BASE CLASSES

CNQueue

DERIVED CLASSES

None

RELATED CLASSES

CNDLList, CNQueueFIFO, CNQueueLIFO, CNQueueSPT, CNPrioQueueFIFO, CNSink, CN-Job, CNStack

DESCRIPTION

`CNQueueRandom` is a queue, implemented as a doubly linked list, that can contain any number (well, sort of ... ;-) CNCL compatible objects. The queueing strategy is Random, that means there is no ordering in the queue.

Constructors:

`CNQueueRandom();`
`CNQueueRandom(CNParam *param);`
`CNQueueRandom(CNRNG *rng);`
: Initialize the Random-queue to an empty state. One may provide a base random number generator (recommended!), otherwise the queue provides its own (default setting: `CNFiboG`.)

In addition to the member functions required by CNCL, `CNQueueRandom` provides:

`virtual bool empty() const;`
: Returns `TRUE`, if the queue is empty.

`virtual bool full() const;`
: Always returns `FALSE`.

`virtual int length() const;`
: Returns the actual queue length.

`virtual void put(CNObject *obj);`
: Puts an object into the queue.

`virtual CNObject *get();`
: Retrieves a random object from the queue.

`virtual CNObject *peek();`
: Retrieves an object from the queue. Unlike `get()`, the object is not removed from the queue.

`virtual void delete_all();`
: Deletes all objects in the queue.

5.13 CNQueueSPT — SPT queue

SYNOPSIS

`#include <CNCL/QueueSPT.h>`

TYPE

`CN_QUEUESPT`

BASE CLASSES

CNQueue

DERIVED CLASSES

None

RELATED CLASSES

CNDLList, CNQueueFIFO, CNQueueLIFO, CNQueueRandom, CNPrioQueueFIFO, CNSink, CNJob, CNStack

DESCRIPTION

`CNQueueSPT` is a queue, implemented as a doubly linked list, that can contain any number (well, sort of ... ;-) CNCL compatible (restrictions see below) objects. The queueing strategy is SPT, that means jobs with the shortest processing time are delivered first. The processing time is read from the public identifier `length` of the `CNJob` object. SPT queues only accept objects derived from `CNJob`.

Constructors:

`CNQueueSPT();`

`CNQueueSPT(CNParam *param);`
: Initialize the SPT-queue to an empty state.

In addition to the member functions required by CNCL, `CNQueueSPT` provides:

`virtual bool empty() const;`
: Returns `TRUE`, if the queue is empty.

`virtual bool full() const;`
: Always returns `FALSE`.

`virtual int length() const;`
: Returns the actual queue length.

`virtual void put(CNObject *obj);`
: Puts an object into the queue. Only objects derived from `CNJob` may be put into a SPT queue.

`virtual CNObject *get();`
: Retrieves the object with the shortest processing time from the queue.

`virtual CNObject *peek();`
: Retrieves the object with the shortest processing time from the queue. Unlike `get()`, the object is not removed from the queue.

`virtual void delete_all();`
: Deletes all objects in the queue.

5.14 CNPrioQueueFIFO — Queue with priority

SYNOPSIS

```
#include <CNCL/PrioQueueFIFO.h>
```

TYPE

`CN_PRIOQUEUEFIFO`

BASE CLASSES

CNQueue

DERIVED CLASSES

None

RELATED CLASSES

CNDLList, CNQueueFIFO, CNQueueLIFO, CNQueueRandom, CNQueueSPT, CNSink, CN-Job, CNStack

DESCRIPTION

`CNPrioQueueFIFO` is a queue, implemented as some FIFO queues, that can contain any number (well, sort of ... ;-) CNCL compatible (restrictions see below) objects. A priority queue consists of a number of simple FIFO queues, one for each priority. The priority value (0 to ...) is taken from the public identifier `priority` of the `CNJob` object. Lower values mean higher priority in access. Only objects derived from `CNJob` are accepted.

Constructors:

`CNPrioQueueFIFO();`
`CNPrioQueueFIFO(CNParam *param);`
`CNPrioQueueFIFO(int prios);`

Initialize the priority-queue to an empty state. The `prios` value determines the number of different priority steps for. By default `prios` is set to two.

In addition to the member functions required by CNCL, `CNPrioQueueFIFO` provides:

`virtual bool empty() const;`
: Returns `TRUE`, if all internal queues are empty.

`virtual bool full() const;`
: Always returns `FALSE`.

`virtual int length() const;`
: Returns the actual queue length, that means the sum of all internal queues.

`virtual void put(CNObject *obj);`
: Puts an object into the queue. Only objects derived from `CNJob` may be put into a priority queue.

`virtual CNObject *get();`
: Retrieves an object from the queue. If there are objects in more than one internal queue the object with the lowest value in `priority` is retrieved.

`virtual CNObject *peek();`
: Retrieves an object from the queue. Unlike `get()`, the object is not removed from the queue.

`virtual void delete_all();`
: Deletes all objects in the queue.

`int priorities();`
: Returns the number of different prioritie steps.

5.15 CNJob — Job object

SYNOPSIS

```
#include <CNCL/Job.h>
```

TYPE

`CN_JOB`

BASE CLASSES

CNObject

DERIVED CLASSES

None

RELATED CLASSES

CNDLList, CNQueueFIFO, CNQueueLIFO, CNQueueRandom, CNQueueSPT, CNPrioQueueFIFO, CNSink, CNJob, CNStack

DESCRIPTION

`CNJob` is a standard job object for the CNCL queues. FIFO, LIFO and Random Queus *can* contain `CNJob`, whereas for SPT and Priority Queues *must* contain it. Users may derive objects from `CNJob` if they need additional data. `CNJob` provides the following public member variables which can be set/read by any user:

```
CNSimTime in;
CNSimTime out;
CNSimTime start;
double orig_length;
double length;
int priority;
```

The three time variables are used to hold the time when a job enters a system, when a job leaves a system and when the serving begins. `length` defines the remaining service time of a job while `orig_length` defines its whole service time. They are equal in case of noninterrupting queueing strategies. `priority` identifies the priority of a job used in priority queues.

Constructors:

`CNJob();`

`CNJob(CNParam *param);`

`CNJob(double len);`

`CNJob(int prio);`

`CNJob(int prio, double len);`
Initialize the Job object, optionally setting length and/or priority.

`CNJob` provides no additional member functions besides the one required by CNCL.

5.16 CNSink — Kitchen Sink

SYNOPSIS

`#include <CNCL/Sink.h.h>`

TYPE

`CN_SINK`

BASE CLASSES

CNQueue

DERIVED CLASSES

None

RELATED CLASSES

CNDLList, CNQueueFIFO, CNQueueLIFO, CNQueueRandom, CNQueueSPT, CNPrioQueue-FIFO, CNJob, CNStack

DESCRIPTION

`CNSink` is a data sink, implemented as queue with only an input side. All objects which are put into this sink are destroyed there. There is no queueing strategy.

Constructors:

`CNSink();`
`CNSink(CNParam *param);`
: Initialize the Sink.

In addition to the member functions required by CNCL, `CNSink` provides:

`virtual bool empty() const;`
: Always returns `TRUE`.

`virtual bool full() const;`
: Always returns `FALSE`.

`virtual int length() const;`
Returns the number of objects deleted in the queue.

`virtual void put(CNObject *obj);`
Puts an object into the queue. It will be deleted there.

`virtual CNObject *get();`
Produces an error. A sink is like a black hole.

`virtual CNObject *peek();`
Produces an error. A sink is like a black hole.

`virtual void delete_all();`
Sets the current sink to its initial state, the number of deleted objects is set to zero.

5.17 CNStack — Stack

SYNOPSIS

```
#include <CNCL/Stack.h>
```

TYPE

`CN_STACK`

BASE CLASSES

CNQueue

DERIVED CLASSES

None

RELATED CLASSES

CNDLList, CNQueueFIFO, CNQueueLIFO, CNQueueRandom, CNQueueSPT, CNPrioQueueFIFO, CNSink, CNJob

DESCRIPTION

`CNStack` is a stack, implemented as a LIFO queue which can hold any number of CNCL compatible objects. It acts very much like a LIFO queue but provides another kind of user interface.

Constructors:

```
CNStack();
CNStack(CNParam *param);
CNStack(long elem);
```

Initialize the Stack, optionally setting the size.

In addition to the member functions required by CNCL, `CNStack` provides:

```
bool empty();
```

Returns `TRUE` when stack is empty.

`long depth() const;`
: Returns the number of objects on the stack.

`void push(CNObject *obj);`
: Pushes an object onto the stack.

`CNObject *pull();`
: Retrieves the last object from the stack.

`CNObject *pop();`
: An alias for `pull()`. Retrieves the last object from the stack.

`void clear();`
: Deletes all objects on the stack.

`long size();`

`void size(long num);`
: Gets/sets the maximum depth of the stack.

6 Event Driven Simulation

CNCL includes a set of classes for performing event driven simulation.

For programmers convinience a script `sim.h` has been added to CNCL, so that all necessary simulation tools are available by typing `#include <sim.h>`. By default the heap scheduler is used, but if the special abilities of `CNEventScheduler` are needed, your program only needs to define `#define NO_HEAP_SCHEDULER`.

6.1 CNEvent — Generic Event

SYNOPSIS

```
#include <CNCL/Event.h>
```

TYPE

`CN_EVENT`

BASE CLASSES

CNDLObject

DERIVED CLASSES

None

RELATED CLASSES

CNEventHandler, CNEventScheduler, CNSimTime

DESCRIPTION

`CNEvent` is a data type for events used in the simulation. Control is passed between different event handlers by sending events to each other. Events have a unique ID field, a priority, a user defined type, an issued and scheduled simulation time, an addressed event handler, a sending event handler, and a pointer to an arbitrary CNCL object.

`CNEvent`s can only be allocated on the heap via `new`.

Constructors:

```
CNEvent();
CNEvent(CNParam *param);
CNEvent(int new_type);
CNEvent(int new_type, const CNSimTime t, int prio=0);
CNEvent(int new_type, CNEventHandler *new_to, const CNSimTime t, CNObject
*new_object=NIL,
int new_prio=0);
```

`CNEvent(int new_type, CNEventHandler *new_from, CNEventHandler *new_to, const CNSimTime t,`
`CNObject *new_object=NIL, int new_prio=0);`

`CNEvent(int new_type, CNEventHandler *new_to, CNObject *new_object=NIL, int new_prio=0);`

Initializes the event, optionally setting the event type, the addressed and sending event handler, the scheduled time, the referenced object, and the priority.

Destructors:

`~CNEvent();`

Deletes the event. The destructor is private and may only be called from `CNEvent` itself or the friend class `CNEventScheduler`. The destructor does NOT delete the referenced object.

In addition to the member functions required by CNCL, `CNEvent` provides:

`static void set_max_events(unsigned long n);`

`static unsigned long get_max_events();`

Gets/Sets maximum number of events. The default value for `max-event` is 100. If `set_max_events()` changes this value, it MUST be called before ANY event is created.

If more than `max-event` events are allocated, a `warning` message is printed and the simulation continues with `max-event` multiplied by 10.

`int priority() const;`

`void priority(int prio);`

`int get_priority() const;`

`void set_priority(int prio);`

Gets/sets the event's priority.

`int type() const;`

`void type(int new_type);`

`int get_type() const;`

`void set_type(int new_type);`

Gets/sets the event's type.

`CNSimTime scheduled() const;`

`void scheduled(const CNSimTime new_scheduled);`

`CNSimTime get_scheduled() const;`

`void set_scheduled(const CNSimTime new_scheduled);`

Gets/sets the event's scheduled simulation time.

`CNSimTime issued() const;`

`CNSimTime get_issued() const;`

Gets the event's issued simulation time.

`CNEventHandler *to() const;`

`void to(CNEventHandler *new_to);`

`CNEventHandler *get_to() const;`

`void set_to(CNEventHandler *new_to);`

Gets/sets the event's addressed event handler.

`CNEventHandler *from() const;`

`void from(CNEventHandler *new_from);`

`CNEventHandler *get_from() const;`

`void set_from(CNEventHandler *new_from);`

Gets/sets the event's sending event handler.

`CNObject *object() const;`

`void object(CNObject *obj);`

`CNObject *get_object() const;`

`void set_object(CNObject *obj);`

Gets/sets the event's referenced object.

`typedef unsigned long CNEventID;`

`CNEventID id() const;`

`CNEventID get_id() const;`

Returns the event's unique ID number.

`static void *operator new(size_t s);`

The `new` operator. It is allocating an event out of a pool

`static void operator delete(void *p);`

The `delete` operator.

`bool after(CNEvent* e2);`

Returns `TRUE` if `*this` Event is scheduled after Event `e2`. If both Events are scheduled at the same time, the priority is checked; after that the Event ID. This method is used by the class `CNEventHeapSched`.

6.2 CNEventExploder — Send Events to multiple EventHandlers

SYNOPSIS

```
#include <CNCL/EventExploder.h>
```

TYPE

`CN_EVENTEXPLODER`

BASE CLASSES

CNEventHandler

DERIVED CLASSES

None

RELATED CLASSES

CNEvent, CNEventHandler, CNEventScheduler, CNSimTime

DESCRIPTION

`CNEventExploder` is a subclass derived from `CNEventHandler`.
It maintains a list of other arbitrary `CNEventHandlers` and forwards all received events to all the `CNEventHandlers` in its list. This can be used to implement "broadcast"-events.

Constructors:

`CNEventExploder();`

`CNEventExploder(CNParam *param);`

Initializes the event exploder.

In addition to the member functions required by CNCL, `CNEventExploder` provides the following member functions:

`virtual void event_handler(const CNEvent *ev);`

This function receives an event and re-sends it to other `CNEventHandlers`.

`virtual void add_handler(CNEventHandler *eh);`
: This function adds a new `CNEventHandler` to the list. It will then receive all events that are send to the `CNEventExploder`.

  If a `CNEventHandler` is added more than once, it will also receive all events as many times.

`virtual void rem_handler(CNEventHandler *eh);`
: This function removes a previously added `CNEventHandler` from the list. It will no longer receive events managed by the `CNEventExploder`.

  If a `CNEventHandler` has been added more than once, it also has to be removed as many times to be completely removed from the `CNEventExploder`.

6.3 CNEventHandler — Abstract Base Class for Event Handlers

SYNOPSIS

`#include <CNCL/EventHandler.h>`

TYPE

`CN_EVENTHANDLER`

BASE CLASSES

CNObject

DERIVED CLASSES

CNEventExploder

RELATED CLASSES

CNEvent, CNEventScheduler, CNSimTime

DESCRIPTION

`CNEventHandler` is a base class for creating simulation event handler. Deriving an event handler class from `CNEventHandler` is necessary for the simulation programm.

Each derived event handler must at least provide the function `event_handler()`, which is defined as pure virtual in class `CNEventHandler`.

Constructors:

`CNEventHandler();`
`CNEventHandler(CNParam *param);`
: Initializes the event handler.

`CNEventHandler(CNStringR name);`
: Initializes the event handler, setting the name to `name`.

In addition to the member functions required by CNCL, `CNEventHandler` provides the following *protected* member functions available to all derived classes:

`virtual void event_handler(const CNEvent *ev) = 0;`
: This function *must* be defined in the derived classes. It should determine what should happen to the transfered Events. Please see the example at the end of this chapter.

`CNEventScheduler *scheduler() const;`
: Returns the scheduler, which called this event handler.

`int state() const;`

`int state(int new_state);`

`int get_state() const;`

`int set_state(int new_state);`
: Gets/sets the event handler's state.

`CNSimTime now() const;`
: Returns the current simulation time.

`CNEventID send_event(CNEvent *ev);`

`CNEventID send(CNEvent *ev);`
: Send an event to the scheduler. Members left uninitialized in the event are set to default values (scheduled time = current time, addressed event handler = this event handler, sending event handler = this event handler). Returns the event's ID.

`CNEventID send_now(CNEvent *ev);`
: Send an event to the scheduler. Same as `send_event()`, but scheduled time is set to the current time. Returns the event's ID.

`CNEventID send_delay(CNEvent *ev, double dt);`
: Send an event to the scheduler. Same as `send_event()`, but scheduled time is set to the current time plus a time delay dt. Returns the event's ID.

`void delete_event(CNEventID id);`
: Deletes event with ID `id` from the scheduler's event list.

`void delete_events(CNEventHandler *evh);`
: Deletes all events from the list that are addressed to event handler `evh`. If `evh` is `NIL`, it deletes all events addressed to this event handler.

6.4 CNEventList — List of Events

SYNOPSIS

`#include <CNCL/EventList.h>`

TYPE

`CN_EVENTLIST`

BASE CLASSES

CNObject

DERIVED CLASSES

None

RELATED CLASSES

CNEvent, CNEventScheduler

DESCRIPTION

`CNEventList` is the list of events managed by `CNEventScheduler`. It is a wrapper around `CNDLList` for creating a list of sorted events. CNEvents are sorted by their scheduled time and their priority.

Constructors:

`CNEventList();`

`CNEventList(CNParam *param);`
: Initializes the event list to empty.

In addition to the member functions required by CNCL, `CNEventList` provides:

`void add_event(CNEvent *ev);`
: Adds an event to the list.

`void delete_event(CNEventID id);`
: Deletes an event from the list, using the ID code.

`void delete_events(CNEventHandler *evh, bool to);`
Deletes all events from the list. If `to` equals TRUE the events addressed to event handler `evh` are deleted, else the events coming from that event handler.

`void delete_all(CNEventID id);`
Deletes all events from the list.

`CNEvent *next_event();`
Gets the next event (the one at the front) from the list.

`CNEvent *peek_event();`

`CNEvent *peek_event(CNEventID id);`
Returns a pointer to the next scheduled event or a pointer to the event with ID `id`, `NIL` if not available.

6.5 CNEventBaseSched — Abstract scheduler base class

SYNOPSIS

`#include <CNCL/EventBaseSched.h>`

TYPE

`CN_EVENTBASESCHED`

BASE CLASSES

CNObjedt

DERIVED CLASSES

None

RELATED CLASSES

None

DESCRIPTION

Constructors:

`CNEventBaseSched();`
`CNEventBaseSched(CNParam *);`
Initializes the `CNEventBaseSched`.

In addition to the member functions required by CNCL, `CNEventBaseSched` provides:

`CNSimTime::time();`
Returns the simulation time.

`CNStatistics *statistics() const;`
`void statistics(CNStatistics *st);`
Sets/Gets the (optional) scheduler statistic. By default no statistic is used.

`void delete_events_from(CNEventHandler *h);`

`void delete_events_to(CNEventHandler *h);`
: Deletes the events coming `from` or adressed `to` the eventhandler `h`.

The following functions are defined virtual and *must* be defined in derived classes:

`void add_event(CNEvent *ev)=0;`

`void send_event(CNEvent *ev)=0;`
: Adds/sends an event to an event handler.

`void delete_event(CNEventID id)=0;`
: Deletes event with ID `id`.

`void delete_events(CNEventHandler *evh, bool to=TRUE)=0;`
: Deletes all events from the list that are addressed to or are comming from event handler `evh`.

`CNEvent *peek_event()=0;`

`CNEvent *peek_event(CNEventID id)=0;`
: Peeks at next event or event with ID `id`. Returns pointer to event, or `NIL` if not available.

`CNEvent *next_event() = 0;`
: Retrieves next event from internal data structure.

`void start();`

`void start(CNEvent *ev);`
: Starts the scheduler with an optional initialization event.

`void stop()=0;`
: Stops the scheduler after processing the current event and deletes all pending events in the event list. May be used inside an event handler to stop the simulation.

`CNEventIterator *create_iterator() = 0;`
: Creates an event iterator.

`void process_events();`
: Process all events.

`void process_now();`
: Processes all events scheduled for the actual simtime..

6.6 CNEventScheduler — Event Scheduler

SYNOPSIS

```
#include <CNCL/EventScheduler.h>
```

TYPE

`CN_EVENTSCHEDULER`

BASE CLASSES

CNEventBaseSched

DERIVED CLASSES

None

RELATED CLASSES

CNEvent, CNEventList, CNEventHandler, CNEventHeapSched

DESCRIPTION

`CNEventScheduler` is the central simulation control. It manages the events and delivers them to the addressed event handlers.

Scheduled events are sorted with respect to their scheduled simulation time and their priority. If there are two or more events with exactly the same scheduled simulation time and the same priority, they are processed in FIFO order.

If this exact behaviour is not strictly required, if scheduled simulation time and priority are sufficient to determine the procession order of events, then CNEventHeapSched should be used instead of CNEventScheduler. CNEventHeapSched avoids some potential performace deficiencies CNEventScheduler might show.

Constructors:

`CNEventScheduler();`
`CNEventScheduler(CNParam *param);`
Initializes the event scheduler.

In addition to the member functions required by CNCL, `CNEventScheduler` provides:

`void add_event(CNEvent *ev);`

`void send_event(CNEvent *ev);`
: Adds/sends an event to an event handler.

`void delete_event(CNEventID id);`
: Deletes event with ID `id`.

`void delete_events(CNEventHandler *evh, bool to=TRUE);`
: Deletes all events from the list that are addressed to event handler `evh` or that are coming from the event handler if `to` equals FALSE.

`CNEvent *peek_event();`

`CNEvent *peek_event(CNEventID id);`

`CNEvent *next_event();`
: Peeks next event or event with ID `id`. Returns pointer to event, or `NIL` if not available.

`void stop();`
: Stops the scheduler after processing the current event and deletes all pending events in the event list. May be used inside an event handler to stop the simulation.

`CNEventIterator *create_iterator();`
: Creates an iterator object to traverse the list of events.

6.7 CNEventHeapSched — Event Scheduler using a heap

SYNOPSIS

```
#include <CNCL/EventHeapSched.h>
```

TYPE

`CN_EVENTHEAPSCHED`

BASE CLASSES

CNEventBaseSched

DERIVED CLASSES

None

RELATED CLASSES

CNEvent, CNEventHandler, CNEventScheduler

DESCRIPTION

`CNEventHeapSched` is a replacement for `CNEventScheduler`. From the user's point of view, it is completely compatible, but it differs in the internally used datastructures and algorithms.

If a high number of events is used simultaneously, the eventlist used in `CNEventScheduler` can become very slow. `CNEventHeapSched` solves that problem by using a more efficient algorithm, a "heap".

However there is one drawback with `CNEventHeapSched`: in contrast to `CNEventScheduler` no FIFO order of processing can be guaranteed if events compare equal. If there are for example two or more events with exactly the same scheduled simulation time and the same priority, then they are processed in random order.

Constructors:

```
CNEventHeapSched();
CNEventHeapSched(CNParam *param);
```

Initializes the event scheduler.

In addition to the member functions required by CNCL, `CNEventHeapSched` provides:

`void add_event(CNEvent *ev);`

`void send_event(CNEvent *ev);`
: Adds/sends an event to an event handler.

`void delete_event(CNEventID id);`
: Deletes event with ID `id`.

`void delete_events(CNEventHandler *evh, bool to=TRUE);`
: Deletes all events from the list that are addressed to or are comming from event handler `evh`.

`CNEvent *peek_event();`

`CNEvent *peek_event(CNEventID id);`
: Peeks at next event or event with ID `id`. Returns pointer to event, or `NIL` if not available.

`CNEvent *next_event();`
: Gets (and removes) the next `CNEvent` from the current heap.

`void stop();`
: Stops the scheduler after processing the current event and deletes all pending events in the event list. May be used inside an event handler to stop the simulation.

`CNEventIterator *create_iterator();`
: Creates an iterator object to traverse the list of events.

6.8 CNEventIterator — iterate through event list

SYNOPSIS

#include <CNCL/EventLIterator.h> #include <CNCL/EventHIterator.h>

TYPE

CN_EVENTLITERATOR CN_EVENTHITERATOR

BASE CLASSES

CNObject

DERIVED CLASSES

None

RELATED CLASSES

CNEvent, CNEventScheduler, CNEventHeapScheduler

DESCRIPTION

The classes CNEventLIterator and CNEventHIterator are used to step through the list of events. Single events may be deleted. The classes can only be created on the heap by calling the member function create_iterator of the current scheduler.

CNEventLIterator and CNEventHIterator provide in addition to the member functions required by CNCL:

CNEvent *next_event();

Returns an event every time it is called. In case the CNEventHeapSched is used it can not be guaranteed that the same event is delivered only once. Furthermore the order in which the events are delivered needn't obey any rule. It is however guaranteed that every event will be returned at least once.

void delete_current_event()

Deletes the event last returned by next_event().

6.9 CNSimTime — Simulation Time

SYNOPSIS

```
#include <CNCL/SimTime.h>
```

TYPE

```
typedef double CNSimTime
```

DESCRIPTION

`CNSimTime` is a double variable which keeps the current simulation time.

6.10 Example of an Event Driven Simulation

The following example shows how to program an M/M/1 queuing system simulation with CNCL events:

```
// -*- C++ -*-

#include <iostream.h>

#include <CNCL/QueueFIFO.h>
#include <CNCL/EventScheduler.h>
#include <CNCL/FiboG.h>
#include <CNCL/NegExp.h>
#include <CNCL/Moments.h>
#include <CNCL/Job.h>

enum { NJOBS=100000 };

enum { EV_JOB, EV_TIMER_G, EV_TIMER_S }; // Event types for M/M/1 simulation

class Server : public CNEventHandler
{
private:
    CNJob *job;             // Served job
    CNQueueFIFO queue;    // CNQueue
    CNRandom &rnd_b;      // Distribution of service time b
    CNMoments t_w, t_b;   // Evaluation tau_w, tau_b
    enum { ST_WAITING, ST_SERVING };

public:
    virtual void event_handler(const CNEvent *ev);

    void print_results();
    void eval_job(CNJob *job);

    Server(CNRandom &rnd) : rnd_b(rnd), job(NIL), t_w("tau_w"), t_b("tau_b")
    {
        state(ST_WAITING);
    }
};

class Generator : public CNEventHandler
{
private:
    CNRandom &rnd_a;      // Distribution of arrival time a
    Server *server;       // Connected queue/server
    long n;

public:
    virtual void event_handler(const CNEvent *ev);
```

```
    Generator(CNRandom &rnd,Server *serv) : rnd_a(rnd),server(serv),n(0) {}
};

void Generator::event_handler(const CNEvent *ev)
{
    if(n == NJOBS)
        // Stop simulation
        return;

    // Incoming event -> generate new Job
    send_now(new CNEvent(EV_JOB, server, new CNJob));
    // Random delay
    send_delay(new CNEvent(EV_TIMER_G), rnd_a());
    n++;
}



void Server::event_handler(const CNEvent *ev)
{
    switch(state())
    {
    case ST_SERVING:
        switch(ev->type())
        {
        case EV_JOB:
            // Incoming job, put into queue
            {
                CNJob *job;
                job = (CNJob *)ev->object();
                job->in = now();
                queue.put(job);
            }
            break;

        case EV_TIMER_S:
            // Timer event, service time run down
            job->out = now();
            // Evaluate job
            eval_job(job);
            delete job;
            job = NIL;
            // Get new job from queue
            if(!queue.empty())
            {
                job = (CNJob *)queue.get();
                job->start = now();
                // Random service time
                send_delay(new CNEvent(EV_TIMER_S), rnd_b());
                state(ST_SERVING);
            }
```

```
            else
                state(ST_WAITING);
            break;
        }
        break;

    case ST_WAITING:
        switch(ev->type())
        {
        case EV_JOB:
            // Incoming job
            job = (CNJob *)ev->object();
            job->in    = now();
            job->start = now();
            // CNRandom service time
            send_delay(new CNEvent(EV_TIMER_S), rnd_b());
            state(ST_SERVING);
            break;
        }
        break;
    }
}

void Server::eval_job(CNJob *job)
{
    t_w.put(job->start - job->in);
    t_b.put(job->out   - job->in);
}


void Server::print_results()
{
    cout << t_w << t_b;
}

main()
{
    CNRNG   *rng   = new CNFiboG;
    CNNegExp rnd_a(10, rng);
    CNNegExp rnd_b( 5, rng);

    Server           server(rnd_b);
    Generator        generator(rnd_a, &server);

    CNEventScheduler scheduler;
    scheduler.start(new CNEvent(EV_TIMER_G, &generator));

    server.print_results();

}
```

7 Array Classes

The class `CNArray` and its derived classes provide arrays of different data types with array range checking. `CNArrayObject` manages pointers to `CNObject`. The other classes `CNArray`<*type*> manage arrays of standard data types.

The main purpose of these classes is to provide arrays with *range checking*, i.e. access to an array element outside the arrays bounds will terminate the program.

Range checking may be disabled by defining the preprocessor macro `NO_RANGE_CHECK`, e.g. by supplying `-DNO_RANGE_CHECK` on the compiler's command line.

Analogously to class `CNArray`, the classes `CNArray2`, `CNArray3` and their derived classes provide 2-dimensional and 3-dimensional arrays of different data types with array range checking.

7.1 CNArray — Abstract Base Class for 1-dimensional Arrays

SYNOPSIS

```
#include <CNCL/Array.h>
```

TYPE

`CN_ARRAY`

BASE CLASSES

CNObject

DERIVED CLASSES

CNArrayObject, CNArrayChar, CNArrayDouble, CNArrayFLoat, CNArrayInt, CNArrayLong

RELATED CLASSES

CNArray2, CNArray3

DESCRIPTION

`CNArray` is the base class of the `CNArray`<*type*> classes. It defines the common interface.

Constructors:

`CNArray();`
`CNArray(size_t xsize);`
    Initializes `CNArray`.

In addition to the member functions required by CNCL, `CNArray` provides:

`size_t get_size() const;`
`size_t size() const;`
    Returns the size of the array.

`void set_size(size_t sz = 0 );`
`virtual void size(size_t sz=0) = 0;`
    Sets the size of the array. The `size`-function must be implemented in the derived classes.

7.2 CNArrayObject — Array of Pointer to CNObject

SYNOPSIS

```
#include <CNCL/ArrayObject.h>
```

TYPE

`CN_ARRAYOBJECT`

BASE CLASSES

CNArray

DERIVED CLASSES

None

RELATED CLASSES

CNArrayChar, CNArrayDouble, CNArrayFLoat, CNArrayInt, CNArrayLong

DESCRIPTION

`CNArrayObject` manages arrays of pointers to `CNObject`.

Constructors:

`CNArrayObject();`

`CNArrayObject(Param *param);`

`CNArrayObject(size_t sz, CNObjPtr def=0);`
: Initializes the array and optionally sets array size to `sz`. All element pointers are initialized to `NIL` or to `def`.

`CNArrayObject(const CNArrayObject &a);`
: Copy constructor.

Destructors:

`~CNArrayObject();`
: Deletes the array. The referenced objects are NOT deleted!

In addition to the member functions required by CNCL, `CNArrayObject` provides:

`typedef CNObject *CNObjPtr;`

`virtual void size(size_t sz = 0 );`
: Sets the size of the array.

`void put (int index, CNObjPtr value);`
: Puts value into array at indexed location.

`CNObjPtr get (int index) const;`
: Returns value of array at indexed location.

`CNObjPtr& operator[] (int index);`
: Access to array by `operator []`.

`CNArrayObject &operator= (const CNArrayObject &a);`
: Defines the `operator =` for the array to allow copying of arrays.

7.3 CNArrayInt — Array of Integer

SYNOPSIS

`#include <CNCL/ArrayInt.h>`

TYPE

`CN_ARRAYINT`

BASE CLASSES

CNArray

DERIVED CLASSES

None

RELATED CLASSES

CNArrayObject, CNArrayChar, CNArrayDouble, CNArrayFLoat, CNArrayLong

DESCRIPTION

`CNArrayInt` manages arrays of integer (the builtin type `int`). `CNArrayInt` is presented here as an example for all `CNArray<`*type*`>` classes. The interface is the same for all classes only considering the different data types.

Constructors:

`CNArrayInt();`

`CNArrayInt(Param *param);`

`CNArrayInt(size_t sz, int def=0);`
: Initializes the array and optionally sets array size to `sz`. All elements are set to the default value `def`.

`CNArrayInt(const CNArrayInt &a);`
: Copy constructor.

Destructors:

`~CNArrayInt();`
Deletes the array.

In addition to the member functions required by CNCL, `CNArrayInt` provides:

`virtual void size(size_t sz = 0 );`
Sets the size of the array.

`void put (int index, int value);`
Puts value into array at indexed location.

`int get (int index) const;`
Returns value of array at indexed location.

`int& operator[] (int index);`
Access to array by `operator []`.

`CNArrayInt &operator= (const CNArrayInt &a);`
Defines the `operator =` for the array to allow copying of arrays.

7.4 CNArray<type> — 1-dimensional Arrays of other <Type>s

DESCRIPTION

CNCL currently provides array classes for the data types `char`, `double`, `float`, `int`, `long`, and `CNObject *` with the classes `CNArrayChar`, `CNArrayDouble`, `CNArrayFloat`, `CNArrayInt`, `CNArrayLong`, and `CNArrayObject` respectively.

All CNCL compatible objects can be stored in a `CNArrayObject`, thus that there is no need for specialized array types.

Nevertheless it is possible to generate arrays of other data types with the `CNarray` script.

Usage:

`CNarray` *name*

The required parameter is the name of the data type. `CNarray` generates two files *ArrayName.h* and *ArrayName.c* with the definition and implementation of the desired array class.

Please note that *name* must be a single word, pointers and references are not allowed, either. If you need an array of pointers or e.g. an array of `unsigned long`, you can use an appropiate typedef:

```
typedef unsigned long ulong;
typedef Data *DataP;
```

and then generate an array

```
CNarray ulong
CNarray DataP
```

yielding the classes `CNArrayUlong` and `CNArrayDataP`.

7.5 CNArray2 — Abstract Base class of 2-dimensional Arrays

SYNOPSIS

`#include <CNCL/Array2.h >`

TYPE

`CN_ARRAY2`

BASE CLASSES

CNObject

DERIVED CLASSES

CNArray2Char, CNArray2Double, CNArray2Float, CNArray2Int, CNArray2Long, CNArray2Object

RELATED CLASSES

CNArray, CNArray3

DESCRIPTION

`CNArray2` is the base class of the `CNArray2<`*type*`>` classes. It defines the common interface.

Constructors:

```
CNArray2();
CNArray2(Param *param);
CNArray2(size_t r, size_t c);
```

Initializes `CNArray2`.The number of rows is r, the number of cols c.

In addition to the member functions required by CNCL, `CNArray2` provides:

```
size_t get_rows() const;
size_t rows() const;
size_t get_cols() const;
```

`size_t cols() const;`
: Returns the number of rows resp. cols.

`virtual void size(size_t r, size_t c) = 0;`

`void set_size(size_t r, size_t c);`
: Resizes the array.

7.6 CNArray2Char — Array of Array of Char

SYNOPSIS

`#include <CNCL/Array2Char.h>`

TYPE

`CN_ARRAY2CHAR`

BASE CLASSES

CNArray2

DERIVED CLASSES

None

RELATED CLASSES

CNArray2Double, CNArray2Float, CNArray2Int, CNArray2Long, CNArray2Object

DESCRIPTION

Constructors:

`CNArray2Char();`

`CNArray2Char(CNParam *param);`

`CNArray2Char(size_t r, size_t c, char def = 0);`
: Initializes the `CNArray2Char` and optionally sets the arraysize to r rows and c cols.

`CNArray2Char(const CNArray2Char &a);`
: Copy constructor.

Destructor:

`~CNArray2Char();`
: Deletes the array.

In addition to the member functions required by CNCL, `CNArray2Char` provides:

`virtual void size(size_t r, size_t c);`
: Resizes the array to r rows and c cols.

`void put(int r, int c, char value);`
: Writes the character value to position (r, c).

`char get(int r, int c) const;`
: Returns the character written on position (r, c).

`CNArrayChar& operator[] (int index);`
: Access to array by `operator []`. The row `index` is returned.

`CNArray2Char &operator= (const CNArray2Char &a);`
: Defines the `operator =` for the array to allow copying of arrays.

7.7 CNArray2<type> — 2-dimensional Arrays of other <Type>s

DESCRIPTION

CNCL currently provides 2 dimensional array classes for the data types `char`, `double`, `float`, `int`, `long`, and `CNObject *` with the classes `CNArray2Char`, `CNArray2Double`, `CNArray2Float`, `CNArray2Int`, `CNArray2Long`, and `CNArray2Object` respectively. The description of those classes is similar to `CNArray2Char`.

All CNCL compatible objects can be stored in a `CNArray2Object`, thus that there is no need for specialized array types.

Nevertheless it is possible to generate arrays of other data types with the `CNarray2` script.

Usage:

`CNarray2` *name*

The required parameter is the name of the data type. `CNarray2` generates two files *Array2Name.h* and *Array2Name.c* with the definition and implementation of the desired array class.

Please note that *name* must be a single word, pointers and references are not allowed, either. If you need an array of pointers or e.g. an array of `unsigned long`, you can use an appropiate typedef:

```
typedef unsigned long ulong;
typedef Data *DataP;
```

and then generate an array

```
CNarray2 ulong
CNarray2 DataP
```

yielding the classes `CNArray2Ulong` and `CNArray2DataP`.

7.8 CNArray3 — Abstract Base class of 3-dimensional Arrays

SYNOPSIS

`#include <CNCL/Array3.h >`

TYPE

`CN_ARRAY3`

BASE CLASSES

CNObject

DERIVED CLASSES

CNArray3Char, CNArray3Double, CNArray3Float, CNArray3Int, CNArray3Long, CNArray3Object

RELATED CLASSES

CNArray, CNArray2

DESCRIPTION

`CNArray3` is the base class of the `CNArray3<`*type*`>` classes. It defines the common interface.

Constructors:

`CNArray3();`
`CNArray3(Param *param);`
`CNArray3(size_t r, size_t c, size_t d);`
Initializes `CNArray3`. The number of rows is `r`, the number of cols `c`, and the depth is set to `d`.

In addition to the member functions required by CNCL, `CNArray3` provides:

`size_t get_rows() const;`
`size_t rows() const;`

`size_t get_cols() const;`

`size_t cols() const;`

`size_t get_depth() const;`

`size_t depth() const;`

Returns the depth of the array and the number of rows resp. cols.

`virtual void size(size_t r, size_t c, size_t d) = 0;`

`void set_size(size_t r, size_t c, size_t d);`

Resizes the array.

7.9 CNArray3Double — Array of Array of Array of Double

SYNOPSIS

```
#include <CNCL/Array3Double.h>
```

TYPE

`CN_ARRAY3DOUBLE`

BASE CLASSES

CNArray3

DERIVED CLASSES

None

RELATED CLASSES

CNArray3Char, CNArray3Float, CNArray3Int, CNArray3Long, CNArray3Object

DESCRIPTION

Constructors:

`CNArray3Double();`

`CNArray3Double(CNParam *param);`

`CNArray3Double(size_t r, size_t c, size_t d, double def = 0);`
: Initializes the `CNArray3Double` and optionally sets the arraysize to `r` rows, `c` cols, and a depth of `d`.

`CNArray3Double(const CNArray3Double &a);`
: Copy constructor.

Destructor:

`~CNArray3Double();`
: Deletes the array.

In addition to the member functions required by CNCL, `CNArray3Double` provides:

`virtual void size(size_t r, size_t c, size_t d);`
: Resizes the array to `r` rows, `c` cols, and a depth of `d`.

`void put(int r, int c, int d, double value);`
: Writes the double value to position (r, c, d).

`double get(int r, int c, int d) const;`
: Returns the double value written on position (r, c, d).

`CNArray2Double& operator[] (int index);`
: Access to array by `operator []`. The row `index` is returned.

`CNArray3Double &operator= (const CNArray3Double &a);`
: Defines the `operator =` for the array to allow copying of arrays.

7.10 CNArray3<type> — 3-dimensional Arrays of other <Type>s

DESCRIPTION

CNCL currently provides 3 dimensional array classes for the data types `char`, `double`, `float`, `int`, `long`, and `CNObject *` with the classes `CNArray3Char`, `CNArray3Double`, `CNArray3Float`, `CNArray3Int`, `CNArray3Long`, and `CNArray3Object` respectively. The description of those classes is similar to `CNArray3Double`.

All CNCL compatible objects can be stored in a `CNArray3Object`, thus that there is no need for specialized array types.

Nevertheless it is possible to generate arrays of other data types with the `CNarray3` script.

Usage:

`CNarray3` *name*

The required parameter is the name of the data type. `CNarray3` generates two files *Array3Name.h* and *Array3Name.c* with the definition and implementation of the desired array class.

Please note that *name* must be a single word, pointers and references are not allowed, either. If you need an array of pointers or e.g. an array of `unsigned long`, you can use an appropiate typedef:

```
typedef unsigned long ulong;
typedef Data *DataP;
```

and then generate an array

```
CNarray3 ulong
CNarray3 DataP
```

yielding the classes `CNArray3Ulong` and `CNArray3DataP`.

8 Object Management

The classes described in this chapter provide facilities necessary for object management, as abstract keys, hash tables, and an object management fronend class.

8.1 CNKey — Abstract Base Class for Object Management via Keys

SYNOPSIS

`#include <CNCL/Key.h>`

TYPE

`CN_KEY`

BASE CLASSES

CNObject

DERIVED CLASSES

CNKeyString, CNKeyInt

RELATED CLASSES

CNHashTable, CNHashStatic, CNHashDynamic, CNHashIterator, CNManager

DESCRIPTION

`CNKey` is an abstract class for managing CNCL compatible objects via keys. Refer to the desription of the classes derived from `CNKey` for further information. Objects of this type can be stored in and retrieved from hash tables.

Constructors:

`CNKey(CNObject *obj = NIL);`
`CNKey(CNParam *param);`
Initializes `CNKey`.

In addition to the member functions required by CNCL, `CNKey` provides:

`void set_object(CNObject *obj);`
`void set_object(CNObject &obj);`
Stores a CNCL compatible object into the key. Normally, an object is stored in a key at creation time via the constructor.

`CNObject *get_object() const;`
: Gets the object stored in the key. If an object is not available, `NIL` is returned.

The following virtual functions are to be defined by derived classes:

`virtual unsigned long hash( unsigned long capacity, int par = 0) const = 0;`
: Function to evaluate the hash-table value.

`virtual bool compare(CNKey *k) const = 0;`

`virtual bool compare(CNKey &k) const = 0;`
: Function to compare two `CNKeys`.

8.2 CNKeyString — Object Management via String Keys

SYNOPSIS

`#include <CNCL/KeyString.h>`

TYPE

`CN_KEYSTRING`

BASE CLASSES

CNKey

DERIVED CLASSES

None

RELATED CLASSES

CNKeyInt, CNHashTable, CNHashStatic, CNHashDynamic, CNHashIterator, CNManager

DESCRIPTION

`CNKeyString` is a class for managing CNCL compatible objects via `CNString` keys. Objects of this type can be stored in and retrieved from hash tables.

Constructors:

`CNKeyString(CNStringR key_string, CNObject *obj = NIL);`
`CNKeyString(CNParam *param);`
: Initializes `CNKeyString`. The supplied string key is used to calculate the hash table position. Therefore, the string key *must* be unique. Make sure, that the string key is valid during the whole lifetime of the respective key. Otherwise operations on this key are unpredictable.

In addition to the member functions required by CNCL, `CNKeyString` provides:

`CNStringR get_key() const;`
: Returns the string key. Unlike the object the string key cannot be changed.

`virtual unsigned long hash( unsigned long capacity, int par = 0) const;`
: Evaluates and returns the hash-table value. `par` is reserved for future use, any other value than zero will result in an fatal error.

`virtual bool compare(CNKey *k) const;`

`virtual bool compare(CNKey &k) const;`
: Compares two `CNKeys`.

8.3 CNKeyInt — Object Management via Integer Keys

SYNOPSIS

`#include <CNCL/KeyInt.h>`

TYPE

`CN_KEYINT`

BASE CLASSES

CNKey

DERIVED CLASSES

None

RELATED CLASSES

CNKeyString, CNKeyObject, CNHashTable, CNHashStatic, CNHashDynamic, CNHashIterator, CNManager

DESCRIPTION

`CNKeyInt` is a class for managing CNCL compatible objects via integer keys. Objects of this type can be stored in and retrieved from hash tables.

Constructors:

`CNKeyInt(unsigned long key_int, CNObject *obj = NIL);`
`CNKeyInt(CNParam *param);`
  Initializes `CNKeyInt`. The supplied integer key is used to calculate the hash table position. Therefore, the integer key *must* be unique.

In addition to the member functions required by CNCL, `CNKeyInt` provides:

`unsigned long get_key() const;`
  Returns the integer key. Unlike the object the integer key cannot be changed.

`virtual unsigned long hash( unsigned long capacity, int par = 0) const;`
Evaluates the hash-table value.

`virtual bool compare(CNKey *k) const;`

`virtual bool compare(CNKey &k) const;`
Compares two `CNKeys`.

8.4 CNKeyObject — Object Management via CNCL object pointers keys

SYNOPSIS

```
#include <CNCL/KeyObject.h>
```

TYPE

`CN_KEYOBJECT`

BASE CLASSES

CNKey

DERIVED CLASSES

None

RELATED CLASSES

CNKeyString, CNKeyInt, CNHashTable, CNHashStatic, CNHashDynamic, CNHashIterator, CNManager

DESCRIPTION

`CNKeyObject` is a class for managing CNCL compatible objects via CNCL object pointer keys. Objects of this type can be stored in and retrieved from hash tables.

Constructors:

`CNKeyObject(CNObject *key_int, CNObject *obj = NIL);`
`CNKeyObject(CNParam *param);`
: Initializes `CNKeyObject`. The supplied object pointer key is used to calculate the hash table position. Therefore, the object pointer key *must* be unique. In normal memory models this is complied.

In addition to the member functions required by CNCL, `CNKeyObject` provides:

`CNObject *get_key() const;`
: Returns the object pointer key. Unlike the object the object pointer key cannot be changed.

`virtual unsigned long hash( unsigned long capacity, int par = 0) const;`
Evaluates the hash-table value.

`virtual bool compare(CNKey *k) const;`

`virtual bool compare(CNKey &k) const;`
Compares two `CNKeys`.

8.5 CNHashTable — Abstract Base Class for Hash Tables

SYNOPSIS

`#include <CNCL/HashTable.h>`

TYPE

`CN_HASHTABLE`

BASE CLASSES

CNObject

DERIVED CLASSES

CNHashStatic, CNHashDynamic

RELATED CLASSES

CNKey, CNKeyString, CNKeyInt, CNHashIterator, CNManager

DESCRIPTION

`CNHashTable` is an abstract base class for storing and retrieving CNCL compatible objects.

Constructors:

`CNHashTable();`
`CNHashTable(CNParam *param);`
: Initializes `CNHashTable`.

`CNHashTable` defines the following structs and constants:

`const unsigned long DEFAULT_HASH_TABLE_CAPACITY = 101;`
: The default capacity of a hash table. This value is used in derived classes.

`struct HashEntry { CNKey *he_CNKey; unsigned long he_HashValue; };`
: Defines a single entry of the hash table.

`CNHashTable` provides the following *virtual* functions which have to be defined in derived classes:

`virtual void store_key(CNKey *k) = 0;`

`virtual void store_key(CNKey &k) = 0;`

Stores a key into the actual hash table (derived from class `HashTable`). The actual hash table is a homogenous table. Therefore, only keys of the *same* type may be stored into one table. If you nevertheless try to store keys of a different type into one table, it might be detected by the methods `get_key()` and `get_object()`. Refer to the description of the derived classes for further information.

`virtual CNKey *get_key(CNKey *k) const = 0;`

`virtual CNKey *get_key(CNKey &k) const = 0;`

Returns the key which matches the supplied key. If no matching key was found, `NIL` is returned.

`virtual CNObject *get_object(CNKey *k) const = 0;`

`virtual CNObject *get_object(CNKey &k) const = 0;`

Returns the object of the key which matches the supplied key. If no matching key was found or if no object has been stored in the key, `NIL` is returned.

`virtual bool reset() = 0;`

Deletes all entries of the actual hash table and resets it to its initial state. This method *does not* free the memory allocated for the keys stored in the hash table or the objects stored in the keys. If the hash table is already empty, `FALSE` is returned, otherwise `TRUE`.

`virtual bool reset_absolutely() = 0;`

Deletes all entries of the actual hash table and resets it to its initial state. This method *frees* the memory allocated for the keys stored in the hash table, but *does not* free memory allocated for objects stored in keys. If the hash table is already empty, `FALSE` is returned, otherwise `TRUE`.

`virtual bool reset_absolutely_w_obj() = 0;`

Deletes all entries of the actual hash table and resets it to its initial state. This method *frees* the memory allocated for the keys stored in the hash table and *frees* the memory for the objects contained in the keys. If the hash table is already empty, `FALSE` is returned, otherwise `TRUE`.

`virtual bool delete_key(Key *k) = 0;`

`virtual bool delete_key(CNKey &k);`

Deletes the key `k` from the actual hash table. This method *neither frees* the memory allocated for the key stored in the hash table *nor* for the object stored in the key. If the supplied key does not match any in the table, `FALSE` is returned, otherwise `TRUE`.

`virtual bool delete_key_absolutely(CNKey *k) = 0;`

`virtual bool delete_key_absolutely(CNKey &k) = 0;`

Deletes the key `k` from the actual hash table. This method *frees* the memory allocated for the key stored in the hash table, but *does not free* the memory allocated for the object stored in the key. If the supplied key does not match any in the table, `FALSE` is returned, otherwise `TRUE`.

`virtual bool delete_key_absolutely_w_obj(CNKey *k) = 0;`

`virtual bool delete_key_absolutely_w_obj(CNKey &k) = 0;`

Deletes the key `k` from the actual hash table. This method *frees* the memory allocated for the key stored in the hash table and *frees* the memory allocated for the object stored in the key. If the supplied key does not match any in the table, `FALSE` is returned, otherwise `TRUE`.

`virtual bool is_full() const = 0;`
: If the actual table's capacity is exhausted, `TRUE` is returned, otherwise `FALSE`.

`virtual bool is_empty() const = 0;`
: If the actual table is empty, `TRUE` is returned, otherwise `FALSE`.

`virtual unsigned long get_num_entries() const = 0;`
: Returns the number of entries of the actual table.

`virtual unsigned long get_capacity() const = 0;`
: Returns the capacity of the current table.

8.6 CNHashStatic — Hash Tables with Static Capacity

SYNOPSIS

```
#include <CNCL/HashStatic.h>
```

TYPE

`CN_HASHSTATIC`

BASE CLASSES

CNHashTable

DERIVED CLASSES

None

RELATED CLASSES

CNHashDynamic, CNKey, CNKeyString, CNHashIterator, CNKeyInt

DESCRIPTION

`CNHashStatic` is a class which provides a hash table with static capacity for storing and retrieving CNCL compatible objects.

Constructors:

`CNHashStatic(unsigned long cap = DEFAULT_HASH_TABLE_CAPACITY);`

`CNHashStatic(CNParam *param);`
: Initializes `CNHashStatic`. The hash table's capacity is set to the value passed to CNHashStatic. The capacity is static, therefore, you cannot change it during the lifetime of an instance of this class.

Destructors:

`~CNHashStatic();`
: Frees all internally allocated resources.

`CNHashStatic` provides the member functions required by CNCL and `CNHashTable`. Some member functions defined in `CNHashTable` and implemented in `CNHashStatic` demand further explanation:

`void store_key(CNKey *k);`
: Stores a key into the homogenous hash table. Only keys of the *same* type may be stored into the same table. The methods `get_key()` and `get_object()` should detect it. If you try to store a key into an already full table, an error message is displayed and the program is terminated.

`bool delete_key(CNKey *k);`
: Deletes the key from the actual hash table which matches the given key. After having deleted a key from the hash table, the whole table is rehashed, i.e. the positions of all entries within the hash table are recalculated and all entries are stored in a new hash table. This might lead to small time delays when handling large hash tables. This method *does not* free the memory allocated for the keys stored in the hash table. If the supplied key does not match any in the table, `FALSE` is returned, otherwise `TRUE`.

`bool delete_key_absolutely(CNKey *k);`
: Deletes the key from the actual hash table which matches the given key. After having deleted a key from the hash table, the whole table is rehashed. This method *does* free the memory allocated for the keys stored in the hash table. If the supplied key does not match any in the table, `FALSE` is returned, otherwise `TRUE`.

The following example shows how to use a `CNHashStatic` object in order to store and retrieve CNCL compatible objects.

```
CNHashStatic tab(200);
CNKeyString ks("Test", NIL);
CNObject *obj = &ks;

tab.store_key(new CNKeyString("Jabba", obj));
tab.store_key(new CNKeyString("Dabba"));
tab.store_key(new CNKeyString("Dooo"));

if (tab.get_object(CNKeyString("Jabba")) != obj)
   cout << "strange behaviour\n";
else
   cout << "found obj\n";

tab.store_key(new CNKeyInt(10, obj)); // error, key of different type

tab.reset_absolutely();

tab.store_key(new CNKeyInt(10, new CNObject)); // okay

// free memory of all keys and their objects
tab.reset_absolutely_w_obj();
```

8.7 CNHashList — Hash Tables with List Collision Management

SYNOPSIS

```
#include <CNCL/HashList.h>
```

TYPE

`CN_HASHLIST`

BASE CLASSES

CNHashTable

DERIVED CLASSES

None

RELATED CLASSES

CNHashDynamic, CNKey, CNKeyString, CNKeyInt

CHHashIterator does NOT work for CNHashList

DESCRIPTION

`CNHashList` is a class which provides a hash table with unrestricted capacity. Contrary to CNHashStatic the keys occuring more than once are stored in lists, i.e. the capacity of the hash table does not restrict the number of keys, that may be stored. Nontheless the capacity should be large enough for the number of keys to be stored in.

Constructors:

`CNHashList(unsigned long cap = DEFAULT_HASH_TABLE_CAPACITY);`
`CNHashList(CNParam *param);`
: Initializes `CNHashList`. The hash table's capacity is set to the value passed to CN-HashList. The capacity is static. Nontheless, you can store more keys since they are stored in lists.

Destructors:

`~CNHashList();`
Frees all internally allocated resources.

`CNHashList` provides the member functions required by CNCL and `CNHashTable`. Some member functions defined in `CNHashTable` and implemented in `CNHashList` demand further explanation:

`void store_key(CNKey *k);`
Stores a key into the homogenous hash table. Only keys of the *same* type may be stored into the same table. The methods `get_key()` and `get_object()` should detect it. If a hashvalue occures the second time it will be stored in a list. Therefore the hash table will never be full unless the memory is exhausted.

`bool delete_key(CNKey *k);`
Deletes the key from the actual hash table which matches the given key. After having deleted a key from the hash table, it is not necessary to rehash the table. This method *does not* free the memory allocated for the keys stored in the hash table. If the supplied key does not match any in the table, `FALSE` is returned, otherwise `TRUE`.

`bool delete_key_absolutely(CNKey *k);`
The same as above. This method *does* free the memory allocated for the keys stored in the hash table. If the supplied key does not match any in the table, `FALSE` is returned, otherwise `TRUE`.

The following example shows how to use a `CNHashList` object in order to store and retrieve CNCL compatible objects.

```
CNHashList tab(200);
CNKeyString ks("Test", NIL);
CNObject *obj = &ks;

tab.store_key(new CNKeyString("Jabba", obj));
tab.store_key(new CNKeyString("Dabba"));
tab.store_key(new CNKeyString("Dooo"));

if (tab.get_object(CNKeyString("Jabba")) != obj)
   cout << "strange behaviour\n";
else
   cout << "found obj\n";

tab.store_key(new CNKeyInt(10, obj)); // error, key of different type

tab.reset_absolutely();

tab.store_key(new CNKeyInt(10, new CNObject)); // okay

// free memory of all keys and their objects
tab.reset_absolutely_w_obj();
```

8.8 CNHashDynamic — Hash Tables with Dynamic Capacity

SYNOPSIS

```
#include <CNCL/HashDynamic.h>
```

TYPE

`CN_HASHDYNAMIC`

BASE CLASSES

CNHashTable

DERIVED CLASSES

None

RELATED CLASSES

CNHashStatic, CNHashIterator, CNKey, CNKeyString, CNKeyInt, CNManager

DESCRIPTION

`CNHashDynamic` is a class which provides a hash table with dynamic capacity for storing and retrieving CNCL compatible objects.

Constructors:

`CNHashDynamic(unsigned long cap = DEFAULT_HASH_TABLE_CAPACITY);`

`CNHashDynamic(CNParam *param);`

Initializes `CNHashDynamic`. The hash table's capacity is set to the value passed to HashDynamic. The capacity is dynamic, i.e. if the number of entries exceeds 3/4 of the hash table's capacity, it is enlarged to a proper value.

Destructors:

`~CNHashDynamic();`

Frees all internally allocated resources.

`CNHashDynamic` provides the member functions required by CNCL and `CNHashTable`. Some member functions defined in `CNHashTable` and implemented in `CNHashDynamic` demand further explanation:

`void store_key(CNKey *k);`
: Stores a key into the homogenous hash table. Therefore, only keys of the *same* type may be stored into the same table. If you nevertheless try to store keys of different types into one table, it might be detected by the methods `get_key()` and `get_object()`. The capacity is dynamic, i.e. if the number of entries exceeds 3/4 of the hash table's capacity, it is enlarged to a proper value.

`bool delete_key(CNKey *k);`
: Deletes the key from the actual hash table which matches the given key. After having deleted a key from the hash table, the whole table is rehashed, i.e. the positions of all entries within the hash table are recalculated and all entries are stored in a new hash table. This might lead to small time delays when handling large hash tables. This method *does not* free the memory allocated for the keys stored in the hash table. If the supplied key does not match any in the table, `FALSE` is returned, otherwise `TRUE`.

`bool delete_key_absolutely(CNKey *k);`
: Deletes the key from the actual hash table, which matches the given key. After having deleted a key from the hash table, the whole table is rehashed. This method *does* free the memory allocated for the keys stored in the hash table. If the supplied key does not match any in the table, `FALSE` is returned, otherwise `TRUE`.

Refer to `CNHashStatic` for an example as to how to use a `CNHashDynamic` object in order to store and retrieve CNCL compatible objects.

8.9 CNHashIterator — Sequential Iterator for Hash Tables

SYNOPSIS

`#include <CNCL/HashIterator.h>`

TYPE

`CN_HASHITERATOR`

BASE CLASSES

CNObject

DERIVED CLASSES

None

RELATED CLASSES

CNHashTable, CNHashDynamic, CNHashStatic, CNKey, CNKeyString, CNKeyInt, CNManager

DESCRIPTION

`CNHashIterator` is an iterator to sequentially traverse a `CNHashTable` hash table.

Constructors:

`CNHashIterator();`

`CNHashIterator(CNParam *param);`
Initializes `CNHashIterator`.

`CNHashIterator(const CNHashTable *new_hash_table);`

`CNHashIterator(const CNHashTable &new_hash_table);`
Initializes `CNHashIterator` with hash table `new_hash_table`. The iterator is reset to the first element of the hash table.

In addition to the member functions required by CNCL, `CNHashIterator` provides:

`void reset(const CNHashTable *new_hash_table);`

`void reset(const CNHashTable &new_hash_table);`

`void reset();`

Resets the iterator to a new hash table `new_hash_table` and/or sets the iterator to the first element of the hash table.

`CNKey *key()`

`CNKey *get_key()`

Gets the referenced key from the current iterator position. It returns the key or `NIL`, if none is available.

`CNKey *first_key();`

`CNKey *first();`

Sets the iterator to the first element of the hash table. It returns the referenced key or `NIL`, if none is available.

`CNKey *last_key();`

`CNKey *last();`

Sets the iterator to the last element of the hash table. It returns the referenced key or `NIL`, if none is available.

`CNKey *next_key();`

`CNKey *next();`

`CNKey *operator ++();`

`CNKey *operator ++(int);`

Moves the iterator to the next element of the hash table. It returns the current referenced key (the one BEFORE moving the iterator) or `NIL`, if none is available.

`CNKey *prev_key();`

`CNKey *prev();`

`CNKey *operator --();`

`CNKey *operator --(int);`

Moves the iterator to the previous element of the hash table. It returns the current referenced key (the one BEFORE moving the iterator) or `NIL`, if none is available.

`CNObject *object()`

`CNObject *get_object()`

Gets the object of the referenced key from the current iterator position. It returns the object or `NIL`, if none is available.

`CNObject *first_object();`

Sets the iterator to the first element of the hash table. It returns the object of the referenced key or `NIL`, if none is available.

`CNObject *last_object();`

Sets the iterator to the last element of the hash table. It returns the object of the referenced key or `NIL`, if none is available.

`CNObject *next_object();`

Moves the iterator to the next element of the hash table. It returns the object of the current referenced key (the one BEFORE moving the iterator) or `NIL`, if none is available.

`CNObject *prev_object();`

Moves the iterator to the previous element of the hash table. It returns the current referenced key (the one BEFORE moving the iterator) or `NIL`, if none is available.

The following examples show how to use a `CNHashIterator` object to traverse an hash table:

Forward:

```
CNHashDynamic hash_table;

...

CNHashIterator trav(hash_table);
CNKey *key;
CNObject *obj;

while(key = trav++)
{
    // Do something with key ...
    obj = key->get_object();
}
```

Alternate forward:

```
for(trav.reset(hash_table); key = trav.key(); trav.next())
{
    // ...
}
```

Forward with objects:

```
for(trav.first(); obj = trav.object(); trav.next_object())
{
    // ...
}
```

Backward:

```
for(trav.last(); key = trav.key(); trav--)
{
    // ...
}
```

The only way to delete all entries of an hash table, that is guaranteed to work with `CNHashIterator`:

```
for(trav.reset(hash_table); key = trav.key(); trav.reset())
{
    // delete object associated with key
    delete key->get_object();
    // delete hash table entry and key
    hash_table->delete_key_absolutely(key);
}
```

8.10 CNManager – Object Management Frontend

SYNOPSIS

`#include <CNCL/Manager.h>`

TYPE

`CN_MANAGER`

BASE CLASSES

CNObject

DERIVED CLASSES

None

RELATED CLASSES

CNHashTable, CNHashDynamic, CNHashIterator, CNKey, CNKeyString, CNKeyInt

DESCRIPTION

`CNManager` is a class which provides facilities for storing and retrieving CNCL compatible objects in a filesystem-like manner. Objects of type `CN_MANAGER` can be compared with directories, whereas all other CNCL compatible objects play the role of the files. `CNManager` internally uses a dynamic hash table for managing the objects.

Constructors:

`CNManager(char *object_name = NIL);`
`CNManager(CNParam *param);`
Initializes `CNManager`.

Destructor:

`~CNManager();`
Frees all internally allocated resources.

In addition to the member functions required by CNCL, `CNManager` provides:

`CNObject *new_object(char *object_name, CNClassDesc desc, CNParam *param = NIL);`
: `new_object()` is used to create a tree of objects similar to a filesystem's tree of directories and files. The class descriptor `desc` determines whether a directory (`CN_MANAGER`) or a file (any other type) is to be created. The `object_name` may consist of a pathname and/or a basename. All directory-names and the basename must be separated by slashes ('/'). The pointer to the new created object is returned, if no error occurs, otherwise, if e. g. the specified pathname was incorrect, `NIL` is returned.

`bool delete_object(char *object_name);`
: Removes the specified object from the internal hashtable. The memory allocated for the object via `new_object()` is deleted, too. As described under `new_object()` you may pass a filesystem-like path to `delete_object()`. If the object is deleted succesfully, `TRUE` is returned, `FALSE` otherwise. If the object to be deleted is a 'directory' and if the 'directory' still contains valid 'files', `delete_object()` fails. You must first delete all subentries before deleting the entry itself.

`CNObject *get_object(char *object_name) const;`
: Returns the object associated to the given pathname. If the specified object does not exist, `NIL` is returned.

`char *get_name();`
: Returns the basename of the respective object.

`bool is_empty() const;`
: Returns TRUE if no object is stored in the storing facilities of this class, FALSE otherwise.

The following example shows how to use `CNManager` objects in order to store and retrieve CNCL compatible objects.

```
// this is the root of the example's object hierarchy
CNManager root;
// some 'directory' pointers
CNManager *mobile, *base, *subbase;
// some 'file' pointers
CNHashDynamic *table1, *table2;

// create a 'directory' in the root 'directory'
mobile = (CNManager *)root.new_object("mobile", CN_MANAGER);
if (mobile == NIL)
    ...
// create another 'directory'
base = (CNManager *)root.new_object("base", CN_MANAGER);
if (base == NIL)
    ...

// now create a 'file' (CNCL compatible object) in the
// 'mobile' 'directory'
table1 = (CNHashDynamic *)mobile->new_object("table1", CN_HASHDYNAMIC);
if (table1 == NIL)
    ...
```

```
// now create a 'subdirectory' of 'base' using an absolute path
subbase = (CNManager *)root.new_object("/base/subbase", CN_MANAGER);
if (subbase == NIL)
    ...

// create a 'file' using a path relative to 'base'
table2 = (CNHashDynamic *)base->new_object("subbase/table2", CN_HASHDYNAMIC);
if (table2 == NIL);
    ...

// now try to get an object
if (base->get_object("subbase/table2") == table2)
    ...

// try to delete 'subbase'
if (!root.delete_object("/base/subbase")) // error, dir not empty
    ...

// first delete all subentries
if (!root.delete_object("/base/subbase/table2"))
    ...

// now delete subbase
if (!base->delete_object("subbase")) // okay
    ...

...
```

9 Miscellaneous Classes

The classes described in this chapter are provided by the CNCL class library for miscellaneous purposes such as:

`common coordinates for the graphical interface to EZD`
`common string handling (as a` *CNObject*`)`
`common integer and double handling (as a` *CNObject*`)`
`support of named object management`
`integer2string and double2string conversion`
`reference counting`
`argument parsing`

9.1 CNCoord — 2-Dimensional Coordinates

SYNOPSIS

```
#include <CNCL/Coord.h>
```

TYPE

`CN_COORD`

BASE CLASSES

CNObject

DERIVED CLASSES

None

RELATED CLASSES

CNICoord

DESCRIPTION

`CNCoord` is a data type for managing 2-dimensional coordinates. It is typically used together with `CNICoord` for world coordinates and pixel coordinates respectively.
A `CNCoord` has `double` x and y members which are public accessible.
`CNCoord`s can be automatically converted to `CNICoord`s and vice versa. This is done by applying the conversion factor `CNCoord::scale`.

Constructors:

```
CNCoord();
CNCoord(CNParam *param);
CNCoord(double vx, double vy);
CNCoord(const CNICoord &v);
CNCoord(const CNCoord &v);
```

Initializes the coordinates object and optionally sets x and y components.

Public accessible members:

`double x;`

`double y;` The x and y components of `CNCoord`.

In addition to the member functions required by CNCL, `CNCoord` provides:

`CNCoord &operator = (const CNCoord &v);`

`CNCoord &operator += (const CNCoord &v);`

`CNCoord &operator -= (const CNCoord &v);`
: Defines the operators `=`, `+=` and `-=` for coordinates by applying the standard C/C++ operators to the x and y components.

The following static member functions are provided to manipulate the conversion scale setting:

`static double CNCoord::get_scale();`

`static double CNCoord::set_scale(double new_scale);`
: Gets/sets the scale setting.

Global operators:

`CNCoord operator + (const CNCoord &a, const CNCoord &b);`

`CNCoord operator - (const CNCoord &a, const CNCoord &b);`
: Adds/substracts coordinates by adding/subtracting the x and y components.

9.2 CNICoord — 2-Dimensional Integer Coordinates

SYNOPSIS

`#include <CNCL/ICoord.h>`

TYPE

`CN_ICOORD`

BASE CLASSES

CNObject

DERIVED CLASSES

None

RELATED CLASSES

CNCoord

DESCRIPTION

`CNICoord` is a data type for managing 2-dimensional integer coordinates. It is typically used with `CNCoord` for pixel coordinates and world coordinates respectively.
A `CNICoord` has `int` x and y members which are public accessible.
`CNICoord`s can be automatically converted to `CNCoord`s and vice versa. This is done by applying the conversion factor `CNCoord::scale`.

Constructors:

```
CNICoord();
CNICoord(CNParam *param);
CNICoord(int vx, int vy);
CNICoord(const CNCoord &v);
CNICoord(const CNICoord &v);
```

Initializes the integer coordinates object and optionally sets x and y components.

Public accessible members:

`int x;`

`int y;` The x and y components of `CNICoord`.

In addition to the member functions required by CNCL, `CNICoord` provides:

`CNICoord &operator += (const CNICoord &v);`

`CNICoord &operator -= (const CNICoord &v);`
Defines the operators `+=` and `-=` for integer coordinates, applying the standard C/C++ operators to the x and y components.

The following static member functions are provided to manipulate the conversion scale setting:

`static double CNICoord::get_scale();`

`static double CNICoord::set_scale(double new_scale);`
Gets/sets the scale setting.

Global operators:

`CNICoord operator + (const CNICoord &a, const CNICoord &b);`

`CNICoord operator - (const CNICoord &a, const CNICoord &b);`
Adds/substracts integer coordinates by adding/subtracting the x and y components.

9.3 CNString — Character String

SYNOPSIS

```
#include <CNCL/String.h>
```

TYPE

`CN_STRING`

BASE CLASSES

CNObject

DERIVED CLASSES

None

RELATED CLASSES

CNNamed

DESCRIPTION

`CNString` is a dynamic string manipulating class. Each `CNString` manages the string itself (stored as a normal C character string), the length of the string, and the allocated space. The characters are indexed from 0 to *length*-1.

Constructors:

```
CNString();
CNString(int extra);
CNString(char c);
CNString(char c, int extra);
CNString(const char* cs);
CNString(const char* cs, int extra);
CNString(const CNString& s);
CNString(const CNString& s, int extra);
```

`CNString(CNParam *param);`
: Initializes the CNString. The `extra` parameter gives a hint as to how many extra characters one expects the string to grow. Its default value is 10.

Destructors:

`~CNString();`
: Deletes the CNString.

In addition to the member functions required by CNCL, `CNString` provides:

`default_extra = 10;`
: Default amount of additional storage allocated. Actually set to 10.

`void resize(unsigned i);`
: Changes the capacity of the string either to its length plus default_extra or to i, whichever is longer.

`void to_lower();`
: Converts all characters in this string to lowercase.

`void to_upper();`
: Converts all characters in this string to uppercase.

`void capitalize();`
: Converts each first character of a word to uppercase, all other characters to lowercase. A new word is either at the beginning of the string or it starts with a space (e.g. "a.R fd" will be capitalized to "A.r Fd").

`void strip_crlf();`
: Removes '\r' and/or '\n' at end of string.

`void strip_lspace();`
`void strip_rspace();`
`void strip_space();`
: Removes the left, the right or all white spaces of the string.

`unsigned capacity() const;`
: Returns the amount of space allocated to this string.

`unsigned length() const;`
: Returns the length of this string.

`CNString after(unsigned pos, unsigned l = 0) const;`
: Returns the l characters of this string after position pos (zero-based) as a string. The character at pos will be the first character of the new string. The default value l = 0 will return all characters after pos.

`CNString before(unsigned pos, unsigned l = 0) const;`
: Returns the l characters of this string before position pos (zero based) as a string. The character at pos will be the last character of the new string. The default value l = 0 will return all characters before pos.

`CNString& add(char c);`
`CNString& add(const char* cs);`
: Adds the character c or the character string cs to the end of this string. If this string does not have enough space allocated it will be changed automatically.

`CNString& del(unsigned pos = 0, unsigned l = 1);`
: Deletes l characters beginning at position `pos` (zero based). Default values are pos = 0 and l = 1 (e.g. a.del() will delete the first character of string a). If l is set to 0 all characters beginning at pos will be deleted (e.g. a.del(0,0) will delete all characters in string a). Negative `pos` values count from end of string.

`CNString& del(char c, int pos = 0);`

`CNString& del(const char* cs, int pos = 0);`

`CNString& del(const CNString& s, int pos = 0);`
: Deletes character c, character string cs or string s at its first occurence in this string after position pos (zero based, default value pos = 0). It deletes nothing if c, cs or s is not in.

`CNString& insert(char c, int pos = 0);`

`CNString& insert(const char* cs, int pos =0);`

`CNString& insert(const CNString& s, int pos = 0);`
: Inserts character c, character string cs or CNString s at position pos (zero based, default value pos = 0).

`CNString& replace(char c, int pos = 0);`
: Replaces the character at position pos (zero based, default value pos = 0) with the character c.

`CNString& replace(const CNString& s, int pos = 0, int l = 0);`
: Replaces the l characters starting at position pos (zero based, default value pos = 0) with the characters of string s. Default value l = 0 will replace as many characters as s.length(). The capacity is changed to the needed space plus the default extra value.

`CNString& replace(char oldc, char newc, int pos = 0);`
: Replaces character oldc at its first occurence after position pos (zero based, default value pos = 0) with newc. If oldc is not in this string after pos nothing will be changed.

`CNString& replace(const CNString& olds, const CNString& news, int pos = 0);`
: Replaces string olds at its first occurence after position pos (zero based, default value pos = 0) with string news. If olds is not in this string nothing will be changed.

`int downsearch(char c, int pos = 0);`

`int downsearch(const CNString& s, int pos = 0);`

`int downsearch(const char *cs, int pos = 0);`
: Returns the last occurence of character c or string s in this string before position pos (zero based, default value pos = 0) or the end of this string if pos = 0 or pos >= len. If c or s is not in this string before pos len is returned.

`int upsearch(char c, int pos = 0);`

`int upsearch(const CNString& s, int pos = 0);`

`int upsearch(const char *s, int pos=0) const;`
: Returns the first occurence of character c or string s in this string after position pos (zero based, default value pos = 0). Returns len if c or s is not in this string after pos.

`bool matches(const char* cs, unsigned pos = 0);`

`bool matches(const CNString& s, unsigned pos = 0);`
: Returns TRUE if the character string cs or the string s is equal to the part of this string starting at position pos (zero based, default value pos = 0).

`operator const char * () const;`
: Returns the the C string component of `CNString` to allow the use of `CNStrings` in a `char *` context.

`char operator ()(int i) const;`

`char &operator [](int i);`

Returns the i-th character (zero based) of this string.

`void operator = (const CNString& s);`

`void operator = (const char* cs);`

`void operator = (char c);`

Replaces this string with a copy of string s, character string cs or character c.

`friend CNString operator + (const CNString& a, const CNString& b);`

Returns a string that is the concatenation of the strings `a` and `b`.

`CNString& operator +=(const CNString &s);`

`CNString& operator +=(const char* cs);`

Appends the C string `s` or the `CNString cs` to the end of this string.

`friend bool operator < (const CNString& a, const CNString& b);`

`friend bool operator > (const CNString& a, const CNString& b);`

`friend bool operator >= (const CNString& a, const CNString& b);`

`friend bool operator <= (const CNString& a, const CNString& b);`

`friend bool operator == (const CNString& a, const CNString& b);`

`friend bool operator != (const CNString& a, const CNString& b);`

Returns `TRUE` if the relation holds between the strings.

`void icopy(istream& strm = cin);`

`istream &operator >> (istream& strm, CNString& s);`

Reads all characters up to (not including) the next newline from input stream strm into the string.

For programming convenience, a type for a `const` reference to a string is provided:

```
typedef const CNString & CNStringR;
```

9.4 CNNamed — CNObject with Name

SYNOPSIS

```
#include <CNCL/Named.h>
```

TYPE

`CN_NAMED`

BASE CLASSES

CNObject

DERIVED CLASSES

None

RELATED CLASSES

CNString

DESCRIPTION

`CNNamed` is a data type for managing CNObject's names in CNString format.

Constructors:

`CNNamed();`
`CNNamed(CNStringR name);`
: Initializes the string name object and optionally sets the name.

In addition to the member functions required by CNCL, `CNNamed` provides:

`CNStringR name() const;`
`CNStringR get_name() const;`
: Returns the object's name.

`void name(CNStringR name) const;`
`void set_name(CNStringR name) const;`
: Sets the object's name.

9.5 CNIniFile — .ini-style config file

SYNOPSIS

```
#include <CNCL/IniFile.h>
```

TYPE

`CN_INIFILE`

BASE CLASSES

CNNamed

DERIVED CLASSES

None

RELATED CLASSES

None

DESCRIPTION

The class `CNIniFile` reads/writes MSDOS-style .INI files.
Example of a .INI file:

```
# This is a comment

[Test]
display = 1
string  = Hallo!
abc
```

In this example the variables display, string and abc of section Test are declared. At this class each single line is handled as a node of a double linked list. Each node has a left entry (variable), a right one (value) and a type entry (COMM, EMPTY, SECTION, ENTRY, ENTRYNOEQ). Depending on the different types, left and/or right entry can be empty.

Attention : This class is still under construction ! !

Constructors:

`CNIniFile();`

`CNIniFile(CNParam *);`

`CNIniFile(CNStringR name);`
: Initializes the `CNIniFile`. The `name` parameter will automatically read the file `name`.INI into double linked list.

In addition to the member functions required by CNCL, `CNIniFile` provides:

`int read()`

`int read(CNStringR name);`
: Reads the specified .ini file. Either the object's name or the `name` parameter is chosen. The whole file is stored as a double linked list.

`int write();`

`int write(CNStringR name);`
: Writes the current list into a .ini file.

`CNStringR get_entry(CNStringR section, CNStringR name, bool first=TRUE);`
: NOT IMPLEMENTED YET.

`CNStringR get_entry(CNStringR name, bool first=TRUE);`
: Returns the entry for `name` as a string.

`double get_double(CNStringR section, CNStringR name, bool first=TRUE);`
: NOT IMPLEMENTED YET.

`double get_double(CNStringR name, bool first=TRUE);`
: Returns the entry for `name` as a double.

`int get_int(CNStringR section, CNStringR name, bool first=TRUE);`
: NOT IMPLEMENTED YET.

`int get_int(CNStringR name, bool first=TRUE);`
: Returns the entry for `name` as an integer.

`bool test_entry(CNStringR section, CNStringR name);`
: NOT IMPLEMENTED YET.

`bool test_entry(CNStringR name, bool first=TRUE);`
: NOT IMPLEMENTED YET.

`CNStringR get_section();`
: Returns the sections's name.

`void set_section(CNStringR section);`
: Sets the internal `CNDLIterator ini_sec` to the begining of the section called `name`

9.6 CNFormInt — Integers as CNStrings

SYNOPSIS

```
#include <CNCL/FormInt.h>
```

TYPE

`CN_FORMINT`

BASE CLASSES

CNString

DERIVED CLASSES

RELATED CLASSES

DESCRIPTION

This class converts Integers to Strings using different kinds of format styles provided by the stream-2.0-implementation. The current formats implemented in this class are 'left' and 'right'. Note: If the Integer needs more characters than the width `w` indicates, the most right ones are ignored.

Constructors:

`CNFormInt();`

`CNFormInt(CNParam *param);`

`CNFormInt(int a, int w);`

`CNFormInt(int a, int w, char fill);`

`CNFormInt(int a, int w, char fill, int f);`

Initializes `CNFormInt`, setting the value to the integer value a (default = 0), the string's width, the fill character fill (default = ' ') and the format f (default = CNFormInt::right)

The different formats implemented in `CNFormInt` are:

`int right = 1`

`int left = 2`

In addition to the member functions required by CNCL, `CNFormInt` provides:

`int get_value();`

`int value();`

Returns the value as an integer.

`void set_value(int a);`

`void value(int a);`

Changes the old String and the old value to a.

`char get_fill();`

`char fill();`

Returns the current fill character.

`void set_fill(char f);`

`void fill(char f);`

Changes the String to the fill character f.

`int get_format();`

`int format();`

Returns the current format as an integer. '1' descibes 'right', '2' describes left.

`void set_format(int f);`

`void format(int f);`

Changes the String to the new CNFormInt::formats f.

`int get_width();`

`int width();`

Returns the String's width as an integer.

`void set_width(int w);`

`void width(int w);`

Changes the old String's width to w.

9.7 CNFormFloat — Doubles as CNStrings

SYNOPSIS

```
#include <CNCL/FormFloat.h>
```

TYPE

`CN_FORMFLOAT`

BASE CLASSES

CNString

DERIVED CLASSES

None

RELATED CLASSES

CNFormInt

DESCRIPTION

This class converts doubles to CNStrings using different kinds of format styles provided by the stream-2.0-implementation. The current formats implemented in this class are 'left' and 'right' in connection with 'scientific', 'showpoint' or 'fixed'. Note: If the number is too long, the right part of it will be cut, no matter wich format style is chosen. If the number is cut (because of the chosen precision) it will be round. If the precision is not in the interval [0,16] problems with the accuracy might occur.

Constructors:

```
CNFormFloat();
CNFormFloat(CNParam *param);
CNFormFloat(double x, int w);
CNFormFloat(double x, int w, char fill);
CNFormFloat(double x, int w, char fill, int format);
CNFormFloat(double x, int w, char fill, int format, int pr);
```

Initializes the `FormFloat` with the value x (default = 0.0), the width w (= 3), the fill character fill (= ' '), the format (= right) and the precision pr (= 6).

The different formats implemented in `CNFormFloat` are:

`int right = 1`

`int left = 2`

`int scientific = 4`

`int showpoint = 8`

`int fixed = 16`

These formats can be devided into two groups: right and left on one hand and scientific, showpoint and fixed on the other. Those two groups can be combiened with each other, e.g. left and scientific (= 6) is as far possible as right and fixed (= 17).

In addition to the member functions required by CNCL, `CNFormFloat` provides:

```
double get_value();
double value();
char get_fill();
char fill();
formats get_format();
formats format();
int get_width();
int width();
int get_precision();
int precision();
```

Returns value, fill character, format style, width or precision of the current CNFormFloat.

```
void set_value(double x);
void value(double x);
void set_fill(char f);
void fill(char f);
void set_format(formats f);
format(formats f);
void set_width(int w);
void width(int w);
void set_precision(int pr);
void precision(int pr);
```

Changes the current CNFormFloat by its value, fill character, format style, width or precision.

9.8 CNInt — Integers derived from CNObject

SYNOPSIS

```
#include <CNCL/Int.h>
```

TYPE

`CN_INT`

BASE CLASSES

CNObject

DERIVED CLASSES

None

RELATED CLASSES

CNDouble

DESCRIPTION

This class provides a long int value derived from CNObject. Such it combines the behavior of the builtin type long int with the possibility to use this type with generic containers (like CNDLList) or as parameters to CNEvents and SDLSignals.
Note: CNInt consumes much more memory than an ordinary long int. Therefore it isn't a good idea to create large arrays of CNInts when one doesn't need its special capabilities.
Constructors:

`CNInt(long n=0);`
`CNInt(CNParam *param);`
Initializes the `Int` with the value n (default = 0).

In addition to the member functions required by CNCL, `CNInt` provides:

`operator long()`
: Conversion of CNInt to ordinary long. This allows the use of `CNInt` in calculations. Note: to explicitly convert a CNInt identifier to a double (or else) write double(long(identifier)). Implicit conversion however works well without casting to long first.

`long operator ++()`

`long operator ++(int)`
: Prefix and postfix version of increment.

`long operator --()`

`long operator --(int)`
: Prefix and postfix version of decrement.

`long operator -()`

`long operator +()`
: Unary minus and plus operator.

`long operator +=(long n)`

`long operator -=(long n)`

`long operator *=(long n)`

`long operator /=(long n)`

`long operator %=(long n)`
: Arithmetic operators where a `CNInt` is on the left and on the right side of an equation.

`long operator ^=(long n)`

`long operator |=(long n)`

`long operator &=(long n)`

`long operator !=(long n)`
: Logical operators where a `CNInt` is on the left and on the right side of an equation.

`long operator <<=(long n)`

`long operator >>=(long n)`
: Left and right shifts of a `CNInt`.

Note: unary * and & operators aren't overloaded.

9.9 CNDouble — Doubles derived from CNObject

SYNOPSIS

```
#include <CNCL/Double.h>
```

TYPE

`CN_DOUBLE`

BASE CLASSES

CNObject

DERIVED CLASSES

None

RELATED CLASSES

CNInt

DESCRIPTION

This class provides a double value derived from CNObject. Such it combines the behavior of the builtin type double with the possibility to use this type with generic containers (like CNDLList) or as parameters to CNEvents and SDLSignals.
Note: CNDouble consumes much more memory than an ordinary double. Therfore it isn't a good idea to create large arrays of CNDoubles when one doesn't need its special capabilities.
Constructors:

`CNDouble(double n=0.0);`
`CNDouble(CNParam *param);`
: Initializes the `Double` with the value n (default = 0.0).

In addition to the member functions required by CNCL, `CNDouble` provides:

`operator double()`

Conversion of CNDouble to ordinary double. This allows the use of `CNDouble` in calculations.
Note: to explicitly convert a CNDouble identifier to an ordinary long (or else) write long(double(identifier)). Implicit conversion however works well without casting to double first.

`double operator -()`

`double operator +()`

Unary minus and plus operator.

`double operator +=(double n)`

`double operator -=(double n)`

`double operator *=(double n)`

`double operator /=(double n)`

Arithmetic operators where a `CNDouble` is on the left and on the right side of an equation.

Note: unary * and & operators aren't overloaded.

9.10 CNGetOpt — Interface to GNU getopt()

SYNOPSIS

`#include <CNCL/GetOpt.h>`

TYPE

`CN_GETOPT`

BASE CLASSES

CNObject

DERIVED CLASSES

None

RELATED CLASSES

None

DESCRIPTION

This class manages the interface to the GNU getopt() functionality. As an example for its use please see the file `tGetOpt.c` at the directory `CNCL/lib/misc/test`.

Constructors:

```
CNGetOpt();
CNGetOpt(CNParam *);
CNGetOpt(int argc, char **argv, char *copts=NIL);
```

In addition to the member functions required by CNCL, `CNGetOpt` provides:

```
enum ParamType { NOPARAM=0, WPARAM=1, OPTPARAM=2 };
```

`struct option { char *name; int has_arg; int *flag; int val; };`
: Describe the long-named options requested by the application. The LONG_OPTIONS argument to getopt_long or getopt_long_only is a vector of 'struct option' terminated by an element containing a name which is zero.

  `has_arg` is NOPARAM (0) if the option does not take an argument, WPARAM (1) if the option requires an argument, or OPTPARAM (2) if the option takes an optional argument.

  If `flag` is not NULL, it points to a variable that is set to the value given in `val` when the option is found, but left unchanged if the option is not found.

  To have a long-named option do something other than set an 'int' to a compiled-in constant, such as set a value from 'optarg', set `flag` to zero and `val` to a nonzero value (the equivalent single-letter option character, if there is one). For long options that have a zero `flag`, `GetOpt` returns the contents of `val`.

`void set_args(int argc, char **argv);`
: Set the `argc` and `argv` options.

`void set_char_options(char *copts);`
: Set the single character option.

`void set_long_options(option *lopts);`
: Set the long options array.

`void add_long_option (char *lopt, ParamType pt, char copt);`
: Adds one long option to the current array (up to a total size of 32 elements). An example for one array entry could be:

```
{ "help", 0, 0, 'h'},   /* Help */
```

`int getopt();`

`int getopt(int argc, char *const *argv, const char *optstring);`

`int operator() ();`
: Return the (short) getopt() value.

`int getopt_long(int argc, char *const *argv, const char *options, const struct option *long_options,`
`int *opt_index);`
: Handles both short and long options.

`int getopt_long_only(int argc, char *const *argv, const char *options, const struct option *long_options,`
`int *opt_index);`
: Handles only long options.

`char *optarg();`

`double optarg_double();`

`int optarg_int()`

`int optind();`

`int optopt();`

`void opterr(int err);`
: These functions return the internal argument, index, unrecognized option character or error value. These values are taken from the original getopt.[h,c] files.

9.11 CNRef — Base class for classes with reference counting

SYNOPSIS

```
#include <CNCL/Ref.h>
```

TYPE

BASE CLASSES

None

DERIVED CLASSES

CNRefObj, CNRefNamed

RELATED CLASSES

CNPtr

DESCRIPTION

This class is the base class for classes with reference counting. Note: `CNRef` is outside of CNCL's inheritance tree and is always used as a further base class of its children, which are also derived from `CNObject`.

With the help of reference counting you can track all references to instances of `CNObject`.

See see Section 9.14 [CNPtr], page 221 and the file `tRef.c` in directory `CNCL/lib/misc/test` for examples.

Constructors:

`CNRef();` Initially the reference counter is set to zero.

`CNRef` provides:

`void ref();`
: Increase the reference counter by one.

`static void ref(CNRef* aRef);`
: Static interface to `ref()`. Calls `aRef->ref()`, if `aRef` is non-`NIL`.

`void deref();`
: Decrease the reference counter by one. If the reference counter already was equal to zero, CNCL aborts with an error message. If the decreased reference counter equals to zero, then the object of this class deletes itself from memory, i.e. `delete this;`! In this case you cannot access this object any longer.

`static void deref(CNRef* aRef);`
: Static interface to `deref()`. Calls `aRef->deref()`, if `aRef` is non-`NIL`.

`unsigned long get_count() const;`
: Returns the number of references.

`static void set_debug(bool r, bool = FALSE);`
: If `r` is set to `TRUE`, all calls to `ref()` and `deref()` are logged and a respective message is output to `cerr`. The second parameter has no funcionality, yet.

`static void decrease(CNRef* aRef);`
: Static function, which decreases the reference counter, if `aRef` is non-`NIL`, and does **not** delete `aRef`, if the reference counter equals to zero.

`static void del(CNRef* aRef);`
: Static function, which decreases the reference counter, if `aRef` is non-`NIL` and the reference counter greater than zero, and which deletes `aRef`, if the reference counter equals to 0.

9.12 CNRefObj — CNObject with reference counting

SYNOPSIS

```
#include <CNCL/RefObj.h>
```

TYPE

`CN_REFOBJ`

BASE CLASSES

CNObject, CNRef

DERIVED CLASSES

None

RELATED CLASSES

CNRefNamed, CNPtr

DESCRIPTION

This class provides a common base for `CNObjects`, all references to which should be kept track of by its second base class `CNRef`.

Constructors:

```
CNRefObj();
```

In addition to the member functions required by CNCL, `CNRefObj` provides no further functions.

9.13 CNRefNamed — CNNamed with reference counting

SYNOPSIS

```
#include <CNCL/RefNamed.h>
```

TYPE

`CN_REFNAMED`

BASE CLASSES

CNNamed, CNRef

DERIVED CLASSES

None

RELATED CLASSES

CNRefObj, CNPtr

DESCRIPTION

This class provides a common base for `CNNamed`s, all references to which should be kept track of by its second base class `CNRef`.

Constructors:

```
CNRefNamed();
CNRefNamed(CNStringR name);
```

In addition to the member functions required by CNCL, `CNRefNamed` provides no further functions.

9.14 CNPtr — Intelligent pointer to CNRefObjs

SYNOPSIS

```
#include <CNCL/Ptr.h>
```

TYPE

`CN_PTR`

BASE CLASSES

CNObject

DERIVED CLASSES

None

RELATED CLASSES

CNRef, CNRefObj, CNRefNamed

DESCRIPTION

This class provides an intelligent pointer to `CNRefObj`s, i.e. if you copy one `CNPtr` to another one, then the reference count of the `CNRefObj` contained in the source `CNPtr` is increased automatically and the reference count of the `CNRefObj` contained in the destination `CNPtr` is decreased automatically. See example below.

Constructors:

```
CNPtr();
CNPtr(CNParam *param);
CNPtr(CNRefObj *o);
CNPtr(const CNPtr &p);
```

In addition to the member functions required by CNCL, `CNPtr` provides:

`CNPtr& operator =(const CNPtr& p)`
: Intelligent assignment operator. Additionally to copying the `CNRefObj` data member, the reference count of the `CNRefObj` data member contained in the source `CNPtr` is increased automatically and the reference count of the `CNRefObj` data member contained in the destination `CNPtr` is decreased automatically.

`CNRefObj *operator ->() const`

`CNRefObj *get_object() const`
: Returns the `CNRefObj` data member.

Example:

```
CNRefObj *source_obj, *dest_obj;
CNPtr    *source, *dest;

source_obj = new CNRefObj;
source_obj->ref();
source = new CNPtr(source_obj);

dest_obj = new CNRefObj;
dest_obj->ref();
dest = new CNPtr(dest_obj);

// ref counts
// source_obj : 1
// dest_obj   : 1

// copy contents of source to dest
// implicitly increase counter of source_obj
// implicitly decrease counter of dest_obj and delete it

*dest = *source;
// ref counts
// source_obj : 2
// dest_obj   : 0 (deleted)

// implicitly decrease counter of source_obj
delete source;
// implicitly decrease counter of source_obj and delete it
delete dest;
```

9.15 CNParseArgs — Argument Parser

SYNOPSIS

```
#include <CNCL/ParseArgs.h>
```

TYPE

`CN_PARSEARGS`

BASE CLASSES

CNObject

DERIVED CLASSES

None

RELATED CLASSES

None

DESCRIPTION

CNParseArgs provides a simple argument parser mainly intended (but not limited) to parse numeric arguments given in the command line of a program.

One enumeration and two additional structures are defined in ParseArgs.h in order to allow easy use of this class:

```
enum BlankHandling {NO_BLANK, BLANK}; // handling of blanks after option

struct OptionDescription {
   char * option_string;    // option string to be parsed
   BlankHandling blank;
   char * default_argument; // default argument
   char * usage;            // optional string for usage
   char * argument_string;  // contains the argument on return
};

struct CNParseArgsList {
```

```
    int number_of_args; // number of elements of args_array
    OptionDescription * args_array; // array containing description
            // of options
    int mandatory_args;     // number of args that have to be given
    int positional_args;    // number of args that may be given without
    // option but in correct positional order
    void (*usage_func)(int argc, char **argv);   // Pointer to a user
    // supplied usage function
};
```

Main features of the class are:

`Default arguments:`
: Default arguments to be used if no argument is specified.

`Usage function:`
: From the information given on the options to be parsed, a usage text can automatically be generated. Thus the user does not need to program any usage function. Nevertheless is it possible, to specify a user supplied usage function. This function can also use the automatic usage function.

`Positional arguments:`
: An arbitrary number of the options can be specified to be positional. I.e. that an argument for this option can be specified without specifying the option. Those arguments are assigned consecutively to the positional options. E.g.: `cnhisto -i infile -o outfile` can be specified as `cnhisto infile outfile` if the first two arguments are specified as positional.

`Blank format:`
: Options can either require a `BLANK` between option and argument or be specified as `NO_BLANK` in which case no blank must be specified. This is usefull e.g. to allow the specification of a printer in the usual manner with the option `-P` like `-Php4m`. It is necessary, that no such option appears as the prefix of any other option. This is checked automatically.

`Option format:`
: Only one kind of option format is recognized, i.e. no distinction is made between long and short option. All options start with a hyphen ('-'). Nevertheless can an option be one or several characters long. A minor disadvantage resulting from this, is that several options, each one consisting of a single character, can't be collected e.g. `-a -e` can not be expressed as `-ae`.

`Numeric format:`
: In order to specify a negative number as an argument to a `BLANK` option an additional hyphen ('-') has to be given in order to distingush the number from an option string. More generally, every argument starting with a hyphen, has to be specified with one additional hyphen. This does not apply to `NO_BLANK` options.

In order to make use of the class an array has to be defined consisting of `OptionDescriptions` containing the option strings to be recognized together with the necessary specifications. E.g.:

```
// Number of arguments to be recognized
const int arg_desc_size = 4;
```

```
OptionDescription arg_desc[arg_desc_size] = {
   {"'option'", BLANK/NO_BLANK, "'default_argument'", "'usage'"}
   {"i", BLANK, NIL, "input file name"},
   {"o", BLANK, "result.dat", "output file name (default 'result.dat'"},
   {"rho", BLANK, "0.15", "Processing time (default 0.15)"}
   {"P", NO_BLANK, "hp4m",
    "Printer (stdout if not specified, default hp4m)"}
};

const int mandatory = 1;
const int positional = 2;

// Forward declartion of 'user_function' to allow specification in
// initialization of 'args'
static void user_function(int argc, char **argv);

CNParseArgsList args = {
    arg_desc_size, arg_desc, mandatory, positional, user_function
};
```

In the example above, the program is designed to take four arguments, the first two being positional, the first one also mandatory, and the fourth one must be specified without blank. Valid invocations of the program include:

```
prog infile -Php4w
prog infile outfile -P
prog -o outfile infile
```

In order to create an object of class `CNParseArgs` an object of struct `CNParseArgsList` has to be defined as shown above. The pointer to the user supplied function `(*usage_func)` may be given as NIL. Then the default usage function is called if neccessary. This default function is accessible to the user as the static member function `static void CNParseArgs::usage_screen(char * name, CNParseArgsList &args)`.

In the main function of the program the object `args` of type `CNParseArgsList` is passed in the call of the constructor of `CNParseArgs`:

```
int main(int argc, char **argv) {
    CNParseArgs parse_args(argc, argv, args);
}
```

The constructor of the object `parse_args` parses the array `argv` and (if the syntax is correct) the address of each argument string given in `argv` is written into `args.args_array[..].argument_string` of the corresponding option.

If an option is not specified at all, the value `NIL` is assigned to the corresponding `[..].argument_string`.

To allow easy access to the argument_strings, the operator '[..]' is overloaded in the class and returns the value of `args.args_array[..].argument_string` .

If an option is given, but no argument, the corresponding `[..].argument_string` holds the address of the string `default_argument` specified in `arg_desc`, i.e. it is set to the specified default.

In this way it is possible, to distinguish whether an option has or has not been specified, e.g.:

```
if (args.args_array[3].argument_string == NIL){
    // option -P not specified
    // print to stdout
} else {
    // print to device given in args.args_array[3].argument_string
    // this will be the default 'hp4m' if only -P was specified
    // without an argument.
}
```

The member function `void CNParseArgs::override_nil(int first_arg, int last_arg = 0)` can be used to assign the default string also if an option has not been specified at all, i.e. if it contains `NIL`, e.g. in the above case:

```
parse_args.override_nil(0,2);
```

Thus the arguments of the first three options in the above example are assigned their default values if they are not specified. If `last_arg` equals '0', only the argument string of option `first_arg` is overriden with the default value (provided that it contained `NIL` before!).

As shown in the example above, a default string can not be specified for a mandatory option and must instead be initialized with `NIL`. This is checked by the constructor.

Syntax Rules

If an option is not specified at all, NIL is returned. If an option is given, but no argument, the default is returned. Thus it is possible to distinguish an option not specified from one specified but without an argument.

If `args.args_array[i].blank` equals `BLANK` the argument has to be separated from the option string by white space characters. If it equals `NO_BLANK` the option must be followed immediately by the argument without any separating character. Such an option must not occur as the prefix of any other argument in the list. This is checked by the program. If the same option string is specified more than once the result is undefined.

A `-` as the first character of an argument has to be given as `--`. This is necessary to allow the specification of negative numbers as arguments. The pointer which is returned in `args.args_array[i].argument_string` is pointing to the second `-`. Thus the argument can directly be parsed semantically!

The first `mandatory_args` arguments of `args.args_array` have to be specified, otherwise the program exits (as long as `usage_func` is `NIL`) and prints a usage text created automatically from the parameters given in `args`. If the pointer `usage_func` is different from `NIL`, that function is called.

The number `positional_args` of arguments may be specified without a preceding option string.

If an argument is specified several times for the same option, the last specification is valid.

Implementation constraints

It is not possible to specify an empty string '`""`' for a positional argument without specifying the option.

Depending on which number of 'mandatory_args' and 'positional_args' is smaller this number of the first arguments is as well mandatory as positional. It is not possible to specify the positional and mandatory property separately for these arguments.

Constructors

`CNParseArgs();`
: Not yet implemented

`CNParseArgs(CNParam *);`
: Not yet implemented

`CNParseArgs (int argc, char **argv, CNParseArgsList &args)`
: Initializes `CNParseArgs`.

In addition to the member functions required by CNCL, `CNParseArgs` provides:

`static void usage_screen(char * prog_name, CNParseArgsList &args);`
: This function is called as appropriate, if the element (*usage_func) in `args` equals NIL. It writes all possible option switches (defined by the user in the instance of `OptionDescription`) together with their meanings and the syntax rules to standard output. The user may also call this function from his own program as it is defined in the public part of the class.

`void override_nil(int first_arg, int last_arg = 0);`
: Those argument strings contained in the instance of `CNParseArgsList` that contain the value `NIL` are set to the default string. This is done starting with element first_arg and ending with element last_arg. If last_arg equals 0 only element first_arg is processed.

`char * operator[](int index);`
: Returns the pointer to the argument string number 'index'.

The following template is also defined in ParseArgs.h to allow an easy parsing of numerical arguments.

```
template <class T> void assign_from_string(const char *string, T &input) {
// Reads input from 'string' and assigns the value to the referent of
```

```
// ‘input’. The string stream object is only created as a temporary
// and will be destructed as the statement is finished.
  istrstream(string, strlen(string)) >> input;
}
```

Example of use:

```
double rho
assign_from_string(args.args_array[2].argument_string, rho);
```

or equivalently:

```
assign_from_string(parse_args[2], rho);
```

10 Unix Classes

The classes described here provide an interface to the UNIX operating system. Thus they will only work on machines where UNIX or compareable systems like LINUX are installed. E.g., You won't be able to use them in a DOS environment.

Additional information about the different Unix systam calls see the man pages and your manual.

10.1 CNPipe — Unix Pipe

SYNOPSIS

```
#include <CNCL/Pipe.h>
```

TYPE

`CN_PIPE`

BASE CLASSES

CNObject

DERIVED CLASSES

None

RELATED CLASSES

None

DESCRIPTION

`CNPipe` creates and manages Unix I/O pipes between two programs, thus that two cooperating processes can transfer data.

Constructors:

```
CNPipe();
CNPipe(CNParam *param);
CNPipe(const CNString& prog);
```

Initializing the pipe. If the program name `prog` is set, the pipe will be opened immediately. Otherwise the open() command has to be used when needed. The destructor closes the pipe if it was not closed before.

In addition to the member functions required by CNCL, `CNPipe` provides:

`int open(const CNString& prog);`

Opens the pipe to the program. If the pipe already exists, the old pipe will be closed before opening the new one. The returned integer value indicates the success of the opening process. If successful 0 is returned, otherwise an error message is shown and -1 is returned.

`int close();`

Closes an existing pipe. If an error occurs, -1 is returned, otherwise a 0.

`ostream & out();`

`istream & in ();`

Returns the input/output stream.

`int fd_in();`

`int fd_out();`

Returns the I/O pipe file descriptors.

`int get_pid();`

Returns the program's PID.

10.2 CNSelect — Class Interface to Select(2) System Call

SYNOPSIS

```
#include <CNCL/Select.h>
```

TYPE

`CN_SELECT`

BASE CLASSES

CNObject

DERIVED CLASSES

None

RELATED CLASSES

None

DESCRIPTION

`CNSelect` is the interface to Unix's "select(2)" system call for synchronous I/O multiplexing. Select(2) (refer to Unix man pages, too) examines the descriptor sets and detects if they are ready for reading, writing, or having an exceptional status.

Constructors:

`CNSelect();`
`CNSelect(CNParam *param);`
`CNSelect(int fd);`
Initializes the three descriptor sets (one for the reading, writing and exceptional descriptor set) to null sets. Optionally the particular descriptor `fd` in the read set.

In addition to the member functions required by CNCL, `CNSelect` provides:

`void add_read(int fd);`

`void del_read(int fd);`

`bool test_read(int fd);`

Adds/deletes/tests file descriptor `fd` to the read set.

`void add_write(int fd);`

`void del_write(int fd);`

`bool test_write(int fd);`

Adds/deletes/tests file descriptor `fd` to the write set.

`void add_except(int fd);`

`void del_except(int fd);`

`bool test_except(int fd);`

Adds/deletes/tests file descriptor `fd` to the except set.

`bool select();`

`bool select(long sec, long usec);`

Returns true if there are any ready descriptors in the three descriptor sets. If the time values `sec` and `usec` for the `timeout` are set, false is returned if the timer expires.

11 EZD Interface Classes

The following classes provide an interface to DEC's Easy Draw application. Easy Draw is an application written in scheme which alows easy drawing on the X 11 Window system. The CNCL simulation programs communicate via UNIX pipes with Easy Draw.
These classes use Xcolors and Xfonts. A list of the colors you can find at `/usr/global/lib/X11` in the file rgb.txt, for a list of fonts use the command `xlsfonts`. Only the color 'clear' is added to allow transparent objects in the drawing.
Additionally to the colors some stipple patterns are defined at EZD. They are called *s0*, *s1*, ... , *s16*, where the number is representing the amount of drawn pixels (in the defined color) in a square of 4-by-4. However, this option is ignored when postscript files are written.

You can find additional information about ezd at the man pages.

11.1 EZD — Base Class for EZD Graphic Objects

SYNOPSIS

```
#include <CNCL/EZD.h>
```

TYPE

`CN_EZD`

BASE CLASSES

CNNamed

DERIVED CLASSES

EZDDrawing, EZDObject, EZDPushButton, ...

RELATED CLASSES

None

DESCRIPTION

`EZD` is the base class for EZD graphic objects. It manages the access to EZD's I/O streams and provides the EZD drawing primitives. For additional information about EZD, refer to the manual pages of EZD.
Constructors:

`EZD();`

`EZD(CNParam *param);`

`EZD(const CNStringR name);`
Initializes and opens the pipe to the EZD process.

In addition to the member functions required by CNCL, `EZD` provides:

`static ostream & out();`
Connects the stream from the program to the EZD process.

`static istream & in();`
: Connects the stream from the EZD process to the program.

`static void draw_point(int x, int y, const CNStringR col)`
: Draws a point at position (x,y) in color `col`.

`static void draw_line(int x1, int y1, int x2, int y2, const CNStringR col, int width = -1);`

`static void draw_dash_line(int x1, int y1, int x2, int y2, const CNStringR col, int width = -1);`
: Draws a (dashed) line from point `(x1,y1)` to point `(x2,y2)` in color `col` and `width` pixels wide. If `width` is <= 0, the minimum of one pixel is used for it.

`static void draw_arc(int x, int y, int w, int h, int a1, int a2, const CNStringR col, int width = -1);`
: Draws an unfilled arc. `(x,y)` are the minimum coordinates of the rectangle defining the arc. The arc's rectangle size is width `w` by height `h`. The arc starts at `a1` degrees and spans `a2` degrees - all angles are measured in the positive mathematical sense, the positive x-axis is the reference axis. The drawing color is `col` and the line width is `width`. If width <= 0 EZD's default value (1 pixel) is used.

`static void draw_fill_arc(int x, int y, int w, int h, int a1, int a2, const CNStringR col);`
: Draws a filled arc (end-point to end-point of the arc). `(x,y)` are the minimum coordinates of the rectangle defining the arc. The arcs rectangle size is width w by heigth h. The arc starts at a1 degrees and spans a2 degrees - all angles measured in positive mathematical sense, the positive x-axis is the reference axis. The drawing color is `col`.

`static void draw_pie_arc(int x, int y, int w, int h, int a1, int a2, const CNStringR col);`
: Draws an arc (the arc is closed by drawing a line from each arc end point to the center of the rectangle defining the arc). `(x,y)` are the minimum coordinates of the rectangle defining the arc. The arcs rectangle size is width w by heigth h. The arc starts at a1 degrees and spans a2 degrees - all angles measured in positive mathematical sense, the positive x-axis is the reference axis. The drawing color is `col`.

`static void draw_rectangle(int x, int y, int w, int h, const CNStringR col, int width = -1);`
: Draws an unfilled rectangle. `(x,y)` are the minimum coordinates. The size is defined by its width w and its heigth h. The drawing color is `col` and the line width is `width`. If width is <= 0 EZD's default value (1 pixel) is used.

`static void draw_fill_rectangle(int x, int y, int w, int h, const`
: CNStringR col);

`static void draw_fill_rectangle(int x, int y, int w, int h, const CNStringR col, CNStringR stipple);`
: Draws a filled rectangle. `(x,y)` are the minimum coordinates. The size is defined by its width w and its heigth h. The drawing color is `col`.

`static void draw_text(int x, int y, const CNStringR text, const CNStringR col, const CNStringR font);`

`static void draw_text(int x, int y, int w, int h, const CNStringR align, const CNStringR text,`
`const CNStringR col, const CNStringR font);`
: Writes `text` in the current drawing. `(x,y)` are the minimum coordinates of the rectangle containing the text. `col` and `font` describe color and font of the text. The optional width w, heigth h and alignment information describe the size of this rectangle and

how the text has to be positioned in it. If the text does not fit in, no text is displayed at all.
NOTE: Alternate functions with const char * args instead of CNStringR are supplied for g++ 2.5.8, because g++ 2.5.8 generates buggy code at least for calls to the 2nd draw_text() with string constants "abc" args. Some of the temporary CNStrings are freed twice.

`static void draw_bitmap(int x, int y, CNStringR filename, CNStringR color1 = "black", CNStringR color2= "");`

`static void draw_bitmap(int x, int y, int w, int h, CNStringR filename, CNStringR color1 = "black", CNStringR color2= "");`

Draws the bitmap for the file `filename` at position `x,y` in width `w` and height `h`. The bits that are on are drawn in `color1`, those which are off in `color2`.
NOTE: The file must contain an X11 bitmap, a monochrome portable bitmap (PBM, type P1), a gray scale portable bitmap (PGM, type P2), or a color portable bitmap (PPM, type P3).
NOTE: Same problem with g++ 2.5.8 as described at the method `draw_text()`.

`static void draw_now();`

Causes any buffered changes to be drawn immideately.

`static void draw_clear();`

Clears the current drawing.

`static void pause(int msec);`

Forces out any buffered changes and stops reading commands from stdin until either `msec` milliseconds pass or some event action issues a quit command.

`static CNString event();`

Reads event from pipe in EZD.

`static bool test_event();`

Returns true if there are ready descriptors in the I/O descriptor sets (see also: CNSelect or man pages to "select(2)" system call)

`static void set_scale(const float xscale_x, const float xscale_y, const int xorigin_x,`
`const int xorigin_y);`

`static int x2pix(const float x);`

`static int y2pix(const float y);`

In some applications it is necessary to map a drawing with a cartesian coordinate system onto a window which uses pixels. `set_scale` specifies the scale factor (default: scale (x,y) = (1.0,1.0), origin (x,y) = (0,0)). `x2pix` and `y2pix` evaluate the coordinate transformation for the given values `(x,y)` by the formula `(value * scale + origin)`, rounded to an integer.

`static void print_window(CNStringR winname, CNStringR dateiname);`

Prints the window named `winname` to disk. The file `dateiname` is in postscript format.

`static void save_drawing();`

`static void restore_drawing();`

Saves/restores the current drawing.

11.2 EZDObject — Interface to EZD Object

SYNOPSIS

```
#include <CNCL/EZDObject>
```

TYPE

CN_EZDOBJECT

BASE CLASSES

EZD

DERIVED CLASSES

EZDQueue, EZDServer, EZDText, EZDTimer

RELATED CLASSES

EZDDrawing, EZDPushButton, EZDWindow

DESCRIPTION

`EZDObject` provides an interface to the objects of Easy Draw. Thus the drawings in all derived classes are handled as objects.
Constructors:

`EZDObject();`
`EZDObject(CNParam *param);`
`EZDObject(int x, int y);`
`EZDObject(const CNStringR name, int x, int y);`
  Initializes the EZD-object with either name "obj" or `&name`. `(x,y)` are the object's minimum coordinates.

Public accessible members:

`int x();` Returns the `EZDObject`'s x coordinate.

`int x(int vx);`
: Returns the `EZDObject`'s old x coordinate and changes it to `vx`.

`int y();` Returns the `EZDObject`'s y coordinate.

`int y(int vy);`
: Returns the `EZDObject`'s old y coordinate and changes it to `vy`.

In addition to the member functions required by CNCL, `EZDObject` provides:

`void start();`

`void start(const CNStringR object_name);`

`void end();`
: "Start/End object drawing" command. All drawing commands between `start()` and `end()` draw the named object in the current drawing. If the named object already exists in the current drawing, it is replaced by the new one. If no drawing commands are specified between `start()` and `end()`, the drawing still exists, but with no graphical representation.

`void delete_obj(const CNStringR draw_name = "draw");`
: Deletes the named drawing in the current object.

`int get_lastxb(void);`

`int get_lastyb(void);`

`int get_lastxe(void);`

`int get_lastye(void);`
: Returns the object's last positions.

`void point(int x, int y, const CNStringR col);`

`void line(int x1, int y1, int x2, int y2, const CNStringR col, int width = -1);`

`void arc(int x, int y, int w, int h, int a1, int a2, const CNStringR col, int width = -1);`

`void fill_arc(int x, int y, int w, int h, int a1, int a2, const CNStringR col);`

`void pie_arc(int x, int y, int w, int h, int a1, int a2, const CNStringR col);`

`void rectangle(int x, int y, int w, int h, const CNStringR col, int width = -1);`

`void fill_rectangle(int x, int y, int w, int h, const CNStringR col);`

`void text(int x, int y, const CNStringR text, const CNStringR col, const CNStringR font);`

`void text(int x, int y, int w, int h, CNString align, CNStringR text, CNStringR col, CNStringR font);`

`void bitmap(int x, int y, int w, int h, CNStringR filename, CNStringR &color1 = "black", CNStringR color2 = "");`

`void bitmap(int x, int y, CNStringR filename, CNStringR color1 = "black", CNStringR color2 = "");`
: Basic drawing commands (exact description see section `EZD` or EZD's man pages). The command's `(x,y)` coordinates determine the position in the object. For the position in the current drawing the object's coordinates `(ox,oy)` are added automatically.

`virtual void redraw();`
: Virtual redraw function (has to be implemented by derived objects).

11.3 EZDDrawing — Interface to EZD Drawings

SYNOPSIS

`#include <CNCL/EZDDrawing.h>`

TYPE

`CN_EZDDRAWING`

BASE CLASSES

EZD

DERIVED CLASSES

None

RELATED CLASSES

EZDObject, EZDPushButton, EZDWindow

DESCRIPTION

`EZDDrawing`
Constructors:

`EZDDrawing();`

`EZDDrawing(CNParam *param);`

`EZDDrawing(const CNString &name);`

`EZDDrawing(const CNString &name, const int x, const int y, const int w, const int h);`
: Sets this drawing to the current drawing. Its name is either `name` or (by default) `"draw"`.

Additional functions :

`void set();`
: sets this drawing to the current drawing.

11.4 EZDPushButton — Interface to EZD Push-Button

SYNOPSIS

```
#include <CNCL/EZDPushButton>
```

TYPE

`CN_EZDPUSHBUTTON`

BASE CLASSES

EZD

DERIVED CLASSES

None

RELATED CLASSES

EZDObject, EZDDrawing, EZDWindow

DESCRIPTION

`EZDPushButton` defines a push-button in the current drawing. The button is defined by it's minimum `(x,y)` coordinates, it's width `w` by heigth `h` rectangle size and it's `CNNamed` object name. The default values are x = y = 0, w = 50, h = 20, name = "button".The button is drawn as a filled white rectangle with a black border. The button's `text` is written in black characters in the center of the button.
When the mouse enters a button with an action, the border is thickened. When the mouse button 1 is pressed, the button colors are reversed. Releasing mouse button 1 the push-button's `action` is taken and it is drawn as before.
NOTE: If you leave the button's area with your mouse during pressing/releasing the mouse button no `action` will be taken.
Constructors:

```
EZDPushButton();
EZDPushButton(CNParam *param);
EZDPushButton(const CNString &name, int vx, int vy, int vw, int vh, const CNString
&vtext);
```

`EZDPushButton(const CNString &name, int vx, int vy, int vw, int vh, const CNString &vtext, const CNString &vaction);`

Initializes the push-button in the current drawing. `name` is the object's name (default: "button"), `(vx, vy)` the minimum (x,y) coordinates (default: (0,0)), `(vw, vh)` the button's width and heigth (50,20), `vtext` the describing text in the current drawing ("button") and `vaction` the action taken when the button is pressed ("log-event").

In addition to the member functions required by CNCL, `EZDPushButton` provides:

`void set_text(const CNString &t);`

Sets the push-button's text to t.

11.5 EZDWindow — Interface to EZD Window

SYNOPSIS

```
#include<CNCL/EZDWindow>
```

TYPE

`CN_EZDWINDOW`

BASE CLASSES

EZD

DERIVED CLASSES

None

RELATED CLASSES

EZDDrawing, EZDObject, EZDPushButton

DESCRIPTION

With EZD a drawing can only be displayed when it is mapped into a window. For this, `EZDWindow` creates named windows and maps drawings into it. The window is always displayed in the upper left corner of the screen and is `w_width` pixels wide and `w_heigth` pixels high. The background color is white and the foreground color is black.
Note: If the named window already exists, the old window is deleted. Windows are only visible if drawings are displayed in them.
Constructors:

```
EZDWindow();
EZDWindow(CNParam *param);
EZDWindow(int w, int h);
EZDWindow(int x, int y, int w, int h);
EZDWindow(const CNString &name, int w, int h);
EZDWindow(const CNString &name, int x, int y, int w, int h);
EZDWindow(const CNString &name, const CNString &title, int w, int h)
```

`EZDWindow(const CNString &name, const CNString &title, int x, int y, int w, int h);`
Creates the window named `name` (default: "win") with the title `title` (default setting equals `name`). The window is `w` pixels wide (default: 200) and `h` pixels high (default: 200). It is positioned at the coordinates `(x,y)` (default: (-1,-1)).

Destructor:

`~EZDWindow();`
Deinitilizes (deletes) the current window.

In addition to the member functions required by CNCL, `EZDWindow` provides:

`int heigth() const;`
`int width() const;`
`int wherex() const;`
`int wherey() const;`
Returns the width, heigth or (x,y) position of the current window.

`void overlay(EZDDrawing *d);`
`void underlay(EZDDrawing *d);`
Overlays/undelays the drawing `d` over the existing drawings in the current window.

`void scale_drawing(EZDDrawing *draw, const int origin_x, const int origin_y,`
`const float d_scale_x, const float d_scale_y, const int scale_linewidth);`
Maps the drawing`draw` with a cartesian coordinate system onto the current window. The `origin_(x,y)` parameters specifie the size of the origin coordinate system, `d_scale_(x,y)` and `scale_linewidth` are the scaling factors.

`void set_zoom(EZDDrawing *, const float factor);`
Zooms the drawing by `factor`.

`void set_norm(EZDDrawing *);`
Sets the drawing back to normalsize.

`void set_auto_resize(const int nargs, EZDDrawing * ...);`
Resizes all drawings of the argument. `nargs` is the number of arguments.

`void delete_drawing(EZDDrawing *d);`
Deltes the drawing `d` in the current window.

`void print_window(const CNString &filename);`
Prints the current window as a postscript file on disk.

11.6 EZDDiagWin — Extra window with x-y diagram

SYNOPSIS

```
#include <CNCL/EZDDiagWin.h>
```

TYPE

CN_EZDDIAGWIN

BASE CLASSES

EZD

DERIVED CLASSES

None

RELATED CLASSES

None

DESCRIPTION

`EZDDiagWin` craetes a named window including a diagram. Actually there are three different styles (as an enum) implemeted in this class:

- DOT : draws the diagram in form of dots
- LINE : draws a line
- HISTO: draws a histogram

Constructors:

```
EZDDiagWin();
EZDDiagWin(CNParam *param);
EZDDiagWin(int w, int h, Style s=LINE);
EZDDiagWin(CNStringR n, int w, int h, Style s=LINE);
EZDDiagWin(CNStringR n, CNStringR title, int w, int h, Style s=LINE);
```

`EZDDiagWin(CNStringR n, CNStringR title, int w, int h, int st, CNStringR cd, CNStringR cb,`
`Style s=LINE);`

Initializes a named `EZDDiagwin`. `title` is the title of the window, `w` and `h` the window's width and height. `st` is the size of a step between two values (default: 1), `cd` is the color of the diagram, `cb` the color of the bar (out-of-range display), `s` the style of the diagram.

In addition to the member functions required by CNCL, `CNEZDDiagWin` provides:

`void style(Style s);`

Changes the style to `s`.

`void add(int v);`

Adds a new value `v` at the current x-position to the diagram.

`void clear();`

Clears the diagram window.

`void set();`

Sets current drawing to diagram window drawing.

11.7 EZDTextWin — ezd window for easy text display

SYNOPSIS

```
#include <CNCL/EZDTextWin.h>
```

TYPE

`CN_EZDTEXTWIN`

BASE CLASSES

EZD

DERIVED CLASSES

None

RELATED CLASSES

None

DESCRIPTION

`EZDTextWin` provides an easy possibility for a text display in an EZD window.
Constructors:

`EZDTextWin(CNParam *param);`

`EZDTextWin(int r=ROWS, int c=COLS, int ix=INCX, int iy=INCY);`

`EZDTextWin(int r, int c, int ix, int iy, CNStringR f);`

`EZDTextWin(CNStringR n, int r=ROWS, int c=COLS, int ix=INCX, int iy=INCY);`

`EZDTextWin(CNStringR n, int r, int c, int ix, int iy, CNStringR f);`

`EZDTextWin(CNStringR n, CNStringR t, int r=ROWS, int c=COLS, int ix=INCX, int iy=INCY);`

`EZDTextWin(CNStringR n, CNStringR t, int r, int c, int ix, int iy, CNStringR f);`

Initializes `EZDTextWin`. `r` is the number of rows, `c` the number of columns. `ix` and `iy` are the x-size of one column / y-size of one row. Thus the total size of the window is (`c` • `ix`) by (`r` • `iy`). The strings are the window's name (`n`), it's title (`t`) and the font (`f`). No text is displayed in the window yet.The constant default values are:
ROWS : 24, COLS : 80, INCX : 9, INCY : 15

In addition to the member functions required by CNCL, `EZDTextWin` provides:

`int width() const;`

`int height() const;`
: Returns the window's width or height.

`int cols() const;`

`int rows() const;`
: Returns the number of rows or cols in the window.

`int incx() const;`

`int incy() const;`
: Returns the x-size of a column or the y-size of a row.

`int row_to_y(int r) const;`

`int col_to_x(int c) const;`
: Returns the y-/x-coordinate of the r-th row/ c-th column.

`void clear();`
: Clears the whole text window, painting everything white.

`void clear(int r, int c, int l);`
: Clears a window area. `r` is the row and `c` the column wich is cleared, (`l` • `incx`) the horizontal length in this area wich is painted white.

`void set();`
: Sets the current drawing to text window drawing.

`void hline(int r, int c, int l);`

`void vline(int r, int c, int l);`
: Draws a vertical/horizontal line of length `l`, begining at the position described by row `r` and col `c`.

`void add(int r, int c, CNStringR s);`

`void add(int r, int c, CNStringR s, CNStringR f);`
: Draws the text `s` in row `r`, column `c`. Possible old text at this position (length of string s) is wiped out. `f` is the font.

11.8 EZDDiag — x-y diagram as an EZDObject

SYNOPSIS

`#include <CNCL/EZDDiag.h>`

TYPE

`CN_EZDDIAG`

BASE CLASSES

EZDObject

DERIVED CLASSES

None

RELATED CLASSES

EZDDiagWin

DESCRIPTION

`EZDDiag` creates a diagram as an ezd object. Actually there are three different styles (as an enum) implemeted in this class:

- DOT : draws the diagram in form of dots
- LINE : draws a line
- HISTO: draws a histogram

Constructors:

```
EZDDiag();
EZDDiag(CNParam *param)
EZDDiag(int w, int h, Style s=LINE);
EZDDiag(int w, int h, int x, int y, Style s=LINE);
EZDDiag(const CNString &name, int w, int h, int x, int y, Style s=LINE);
```

`EZDDiag(int w, int h, int x, int y, Style s, CNString cd, CNString cb);`

`EZDDiag(const CNString &name, int w, int h, int x, int y, Style s, CNString cd, CNString cb);`

Initializes `EZDDiag` at the position `(x,y)`, default (0,0), of the current drawing as a named ezd object. `w` (default: 200) and `h` (100) are the object's width and heigth. `cd` ("black") and `cb` ("red") are color-settings for the diagram and the bar which is drawn if the value is out of the given range - e.g. negative values or values `>` 100 at the default setting. The possible styles `s` are described above. The object is drawn frameless, but a frame can be added, see the functions below.

In addition to the member functions required by CNCL, `EZDDiag` provides:

`void style(Style s);`
: Sets the style of the diagram to `s`.

`void set_color_draw(CNString c);`
: Sets the drawing color to `c`.

`void set_color_bar(CNString c);`
: Sets the color of the bar (out-of-range display) to `c`.

`void add(int v);`
: Adds the value `v` to the diagram.

`void clear();`
: Clears the diagram.

`virtual void redraw();`
: Redraws the diagram.

`void set_frame();`
: Sets a frame to the diagram.

11.9 EZDBlock — Block with small rectangles for bit display

SYNOPSIS

`#include <CNCL/EZDBlock.h>`

TYPE

`CN_EZDBLOCK`

BASE CLASSES

EZDObject

DERIVED CLASSES

None

RELATED CLASSES

None

DESCRIPTION

`EZDBlock` draws a block of several square rectangles. Each rectangle's side is `bsize` long, so that the total size of the block is determined by the number of rows and cols. Each rectangle's color is representing the state of a bit, `black` if the bit is turned on and `white` if turned off. The frame of all rectangles is 1 pixel by default.
Constructors:

`EZDBlock();`

`EZDBlock(CNParam *param);`

`EZDBlock(const CNString &name, int x, int y, int r, int c, int b);`

Initializes the `EZDBlock` (does not draw it yet). `&name` is the `EZDObjects` name, `(x,y)` its minimum coordinates (default (0,0)). The block is made out of r by c square rectangles, each side b long. At the initialzation all bits are assumed to be set off (drawn white).

In addition to the member functions required by CNCL, `EZDBlock` provides:

`virtual void redraw();`
Draws / redraws the whole block.

`void on(int b);`

`void off(int b);`
Turns bit on(black)/off(white) and sets the color for it; redraws the (whole) block.

11.10 EZDPopUp — Interface to EZD popup menu

SYNOPSIS

```
#include <CNCL/EZDPopUp.h>
```

TYPE

`CN_EZDPOPUP`

BASE CLASSES

EZDObject

DERIVED CLASSES

None

RELATED CLASSES

None

DESCRIPTION

`EZDPopUp` is the interface to ezd's popup menus. It defines a named scheme that is called as the action on a button down event to display a popup menu. The scheme's functions, displayed names and all other parameters are stored in the struct `MENU`. When button 1 is pressed down, the menu comes up. Releasing the button executes the action associated with the current menu entry. Moving the mouse outside the menu without releasing the button will cause the menu to disappear, no action is executed. For additional information see the ezd manual.
Constructors:

`EZDPopUp();`
`EZDPopUp(CNParam *param);`
`EZDPopUp(const int nmenu, const MENU *menu ...);`
Initializes the conection to ezd's popup with a variable size of parameters. `nmenu` is the total number of menu items, `MENU` a struct descibed below.

In addition to the member functions required by CNCL, `EZDPopUp` provides:

`typedef struct menue MENU;`

`MENU` is a struct of :

`int posx, posy`
: minimum (x,y) coordinates of the menu

`int height`
: height of the menu

`CNString title`
: menu title, used as name of the scheme

`CNString t_color`
: title color

`CNString b_color, f_color`
: color of the background and the frame

`CNString* inhalt`
: menu entries

`CNString* scheme_func`
: scheme functions

`int anz`
: total number of menu entries

Take note that the [i]th entrie and the [i]th function are associated with each other. If `anz` is larger than the correct number the program will terminate, if it is smaller, all entries after that number are set inactive.

11.11 EZDQueue — Graphical Representation of a Queue

SYNOPSIS

```
#include <CNCL/EZDQueue.h>
```

TYPE

CN_EZDQUEUE

BASE CLASSES

EZDObject

DERIVED CLASSES

None

RELATED CLASSES

EZDServer

DESCRIPTION:

`EZDQueue` is the graphical representation of a queue. This drawing is handled as an `EZDObject`, in which x is the most left and y the middle axis' coordinate. The bar, itself an `EZDObject`, is representing the amount of tasks waiting inside the queue.
Constructors:

`EZDQueue();`

`EZDQueue(Param *param);`

`EZDQueue(int x, int y, int width = WIDTH, int heigth = HEIGTH, int tail = TAIL);`

`EZDQueue(const CNString &name, int x, int y, int width = WIDTH, int heigth = HEIGTH, int tail = TAIL);`

Initializes an `EZDQueue` as an `EZDObject` (default object name : "queue") inside the current drawing at the coordinates `(x,y)` (default value (0,0)). Width (100 pixels) and heigth (20 pixels) are the queue body's rectangle size measured in pixels, tail (20 pixels) the tail's length.

In addition to the member functions required by CNCL, `EZDQueue` provides:

`int left() const;`

`int rigth() const;`

`int upper() const;`

`int lower() const;`
: Returns the left, right, upper or lower coordinate of the object queue inside the current drawing.

`virtual void redraw();`
: Redraws this queue in the current drawing.

`int length() const;`

`int get_length() const;`

`void length(int l);`

`void set_length(int l);`
: Get/set length of the queue-bar.

`void color(const CNString &c);`
: Sets the color of the queue-bar to `c` and redraws the queue.

11.12 EZDServer — Graphical Representation of a Server

SYNOPSIS

```
#include <CNCL/EZDServer.h>
```

TYPE

CN_EZDSERVER

BASE CLASSES

EZDObject

DERIVED CLASSES

None

RELATED CLASSES

EZDQueue

DESCRIPTION:

`EZDServer` is the graphical representation of a server. This drawing is handled as an `EZDObject`, in which x is the most left and y the middle axis' coordinate.
Constructors:

`EZDServer();`

`EZDServer(CNParan *param);`

`EZDServer(int x, int y , int radius = RADIUS);`

`EZDServer(const CNString &name, int x, int y, int radius=RADIUS);`
: Initializes an `EZDServer` as an `EZDObject` (default object name : "server") inside the current drawing at the coordinates `(x,y)` (default value (0,0)) with `radius` (default 20 pixels).

In addition to the member functions required by CNCL, `EZDServer` provides:

`virtual void redraw();`
: Redraws the server in the current drawing.

`void color(const CNString &c);`
: Sets the color of the server to `c` and redraws the server in the current drawing.

11.13 EZDText — EZD Object with Text

SYNOPSIS

```
#include <CNCL/EZDText.h>
```

TYPE

CN_EZDTEXT

BASE CLASSES

EZDObject

DERIVED CLASSES

None

RELATED CLASSES

None

DESCRIPTION:

`EZDText` writes text into a clear rectangle. The rectangle is handled as an `EZDObject`, its size is automatically as large as necessary to hold the text in the given `font` and `(x,y)` are its minimum coordinates in the current drawing. Note that the `font` name is specified as a string in X font name terminology.
Constructors:

```
EZDText();
EZDText(CNParam *param);
EZDText(int x, int y, const CNString &text);
EZDText(const CNString &name, int x, int y, const CNString &text);
EZDText(int x, int y, const CNString &text, const CNString &font);
EZDText(const CNString &name, int x, int y, const CNString &text, const CNString
&font);
```

Initializes the `EZDObject` and displays the `text` in the current drawing at the `(x,y)` coordinates. If no `EZDObject` name is chosen it is called "text".

In addition to the member functions required by CNCL, `EZDText` provides:

`virtual void redraw();`
: Redraws the `text` in the current drawing.

`void set_text(const CNString &t);`
`void set_text_val(int x);`
`void set_text_val(const CNString &t, int x);`
`void set_text_val(double x);`
`void set_text_val(const CNString &t, double x);`
: Creates `text` out of text, integer or double and displays it in the current drawing.

11.14 EZDTimer — Graphical Representation of a Timer

SYNOPSIS

```
#include <CNCL/EZDTimer.h>
```

TYPE

CN_EZDTIMER

BASE CLASSES

EZDObject

DERIVED CLASSES

None

RELATED CLASSES

None

DESCRIPTION

`EZDTimer` draws an `EZDObject` representing a timer. It is realized as a filled pie arc inside of an oval. The `(x,y)` coordinates are the center coordinates of the object in the current drawing. The timer starts (by default) at 270 degrees (12 o'clock position) as an offset value. The pie dimension angle (in degrees) is the pie's size. The default colors are black on white backround. Please note that the timer doesn't count automatically, each change of the shown value has to be programmed.
Constructors:

`EZDTimer();`

`EZDTimer(CNParam *param);`

`EZDTimer(int x, int y, int radiusx=RADIUSX, int radiusy=RADIUSY);`

`EZDTimer(const CNString &name, int x, int y, int radiusx=RADIUSX, int radiusy=RADIUSY);`

Initializes the timer as an `EZDObject`. The default object's name is "Timer". Radiusx and radiusy are the oval's radius in (x,y) direction, measured in pixels (default:(5,5))

In addition to the member functions required by CNCL, `EZDTimer` provides:

`int left();`
`int right();`
`int upper();`
`int lower();`
: Returns the oval's most left/right/upper/lower layout coordinates in pixels.

`virtual void redraw();`
: Redraws the timer in the current drawing.

`int offset_angle()`
`int get_offset_angle()`
: Returns the timer's current offset angle.

`void offset_angle(int a);`
`void set_offset_angle(int a);`
: Sets the timer's offset angle to `a` degrees and redraws the pie.

`int angle();`
`int get_angle();`
: Returns the timer's pie dimension angle.

`void angle(int a);`
`void set_angle(int a);`
: Sets the timer's pie dimension angle to `a` degrees and redraws the pie.

`void color(const CNString &c);`
: Sets the timer's color to `c` and redraws the timer.

`void activate();`
`void deactivate();`
: Activates/deactivates an additional oval in 4 pixels distance to the timer.

`void active_color(const CNString &c);`
: Sets the additional oval's color to `c` but does not redraw it.

12 Fuzzy classes

CNCL provides a set of classes for building fuzzy inference engines. Available are fuzzy sets, fuzzy variables, and a inference engine based on fuzzy rules. The membership values are normalized (by default), but can be changed to max and min values in different classes. All fuzzy sets and functions are realized with crisply defined membership values (type 1 fuzzy sets and functions). Currently prod-min inference is used and a center-of-gravity output defuzzification. This will be extended in a future release, providing different operators for aggregation, inference, and accumulation. Up to now two different fuzzy set representations are implemented: LR-representation and arrays.

For more information about fuzzy logic see:
Zimmermann,. H.-J. [1991]. Fuzzy Set Theory And Its Applications. Kluwer Accademic Publishers.
Additionally, the FAQ of the newsgroup "comp.ai.fuzzy" should be recommended in this context.

12.1 CNFClause — Clause of a fuzzy rule

SYNOPSIS

```
#include <CNCL/FClause.h>
```

TYPE

CN_FCLAUSE

BASE CLASSES

CNObject

DERIVED CLASSES

None

RELATED CLASSES

DESCRIPTION

A `CNFClause` is basically a structure containing a fuzzy variable / fuzzy value pair. The members

```
CNFVar *var; // Linguistic variable
CNFSet *value; // Linguistic value
```

are public accessible.

Constructors:

`CNFClause();`
`CNFClause(CNParam *param);`
`CNFClause(CNFVar *xvar, CNFSet *xset);`
`CNFClause(CNFVar &xvar, CNFSet &xset);`
    Initializes a `CNFVar` and a `CNFSet` to a `CNFClause`.

In addition to the member functions required by CNCL, `CNFClause` provides:

`void print_clause(ostream &strm, bool lhs) const;`
    Prints the public accessible variables on the output stream `&strm`.

12.2 CNFVar — Fuzzy variable

SYNOPSIS

`#include <CNCL/FVar.h>`

TYPE

`CN_FVAR`

BASE CLASSES

CNNamed

DERIVED CLASSES

None

RELATED CLASSES

None

DESCRIPTION

This class creates a fuzzy variable and its corrosponding linguistic variable. For example, the linguistic variable "temperature" migth be programmed this way:
CNFVar temp("Temperature", 0, 100);
enum { COLD, MEDIUM, WARM, HOT };
temp.add_value_set(COLD, new CNFSetTrapez("cold", 0, 10, 0, 10));
temp.add_value_set(MEDIUM, new CNFSetTrapez("medium", 20, 20, 10, 10));
temp.add_value_set(WARM, new CNFSetTrapez("warm", 30, 40, 10, 5));
temp.add_value_set(HOT, new CNFSetTrapez("hot", 40,100, 10, 0));
Here trapezium functions have been assumed as membership function, but all other kinds of (implemented) functions are possible.

Constructors:

`CNFVar(CNParam *param);`

`CNFVar(double min = 0, double max = 1);`

`CNFVar(const CNString &xname, double min = 0, double max = 1);`

Initializes the `CNFVar`.

In addition to the member functions required by CNCL, `CNFVar` provides:

`double value( double x );`

`double set_value( double x );`
: Sets the (non fuzzy) variable value to `x` and returns the old value.

`double value() const`

`double get_value()const;`
: Returns the (non-fuzzy) value.

`CNFSet* fuzzy_value( CNFSet* x );`

`CNFSet* set_fuzzy_value( CNFSet* x );`
: Sets the fuzzy variable value (fuzzy set) to `x` and returns the old value.

`CNFSet* fuzzy_value() const;`

`CNFSet* get_fuzzy_value() const;`
: Returns the fuzzy value (fuzzy set).

`double xmin() const;`

`double get_xmin() const;`

`double xmax() const;`

`double get_xmax() const;`
: Returns the minimum and maximum values of the variable's range.

`void add_value_set(CNFSet &fset);`

`void add_value_set(CNFSet *fset);`

`void add_value_set(int i, CNFSet &fset);`

`void add_value_set(int i, CNFSet *fset);`
: Adds the fuzzy set `fset` to the array of affiliated sets, either at position `i` (overwritting) or at the end (extending).

`CNFSet *get_value_set(int i);`
: Returns the fuzzy set at array position `i`.

`double get_membership( int i);`
: Gets the membership value of the fuzzy set at array position `i`.

`void print_membership();`
: Computes the membership value for all fuzzy sets using the current value and prints the result.

12.3 CNFRule — Fuzzy Rule

SYNOPSIS

`#include <CNCL/FRule.h>`

TYPE

`CN_FRULE`

BASE CLASSES

CNObject

DERIVED CLASSES

None

RELATED CLASSES

None

DESCRIPTION

Constructors:

`CNFRule();`
`CNFRule(CNParam *param);`
    Initializes a `CNFRule` variable.

In addition to the member functions required by CNCL, `CNFRule` provides:

`void add_lhs(CNFClause *clause);`
`void add_lhs(CNFClause &clause);`
`void add_rhs(CNFClause *clause);`
`void add_rhs(CNFClause &clause);`
    Adds a `CNFClause` to the LHS or the RHS rules.

`int get_n_lhs() const;`

`int get_n_rhs() const;`
: Gets the total number of LHS/RHS clauses.

`CNFClause* get_lhs(int i) const;`

`CNFClause* get_rhs(int i) const;`
: Returns the LHS/RHS clause from array-position `i`.

`double certainty(double x);`

`double set_certainty(double x);`
: Sets the certainty to `x` and returns the old certainty.

`double certainty() const;`

`double get_certainty() const;`
: Returns the certainty.

`double aggregate_value() const;`
: Returns the current value of the aggregation.

`double aggregate()`
: Aggregates the LHS rule inputs and returns the result.

12.4 CNFRuleBase — Rule base and Fuzzy inference engine

SYNOPSIS

```
#include <CNCL/FRuleBase.h>
```

TYPE

`CN_FRULEBASE`

BASE CLASSES

CNNamed

DERIVED CLASSES

None

RELATED CLASSES

None

DESCRIPTION

Constructors:

`CNFRuleBase();`
`CNFRuleBase(CNParam *param);`
`CNFRuleBase(CNStringR name);`
: Initializes the `CNFRuleBase` with `name` as the object's name.

In addition to the member functions required by CNCL, `CNFRuleBase` provides:

`void add_rule(CNFRule *rule);`
`void add_rule(CNFRule &rule);`
: Adds rule to base.

`void add_in_var(CNFVar *rule);`
`void add_in_var(CNFVar &rule);`

`void add_out_var(CNFVar *rule);`

`void add_out_var(CNFVar &rule);`

Adds input/output variables to the base.

`int get_n_rules()const;`

`int get_n_in_vars()const;`

`int get_n_out_vars()const;`

Returns the number of rules/input variables/output variables in this base.

`int resolution()const;`

Returns the output fuzzy set resolution.

`void inference(CNFVar *var, CNFSet *set , double match, CNFSetArray &res);`

Computes the inference set for variable `var` and value `set`, using value `match` of LHS aggregation. The result is stored in the array fuzzy set `res`.

`void evaluate(CNFVar *var, CNFSetArray &res);`

Computes output set for variable `var`, combining the `inference()` results of all rules. The result is stored in the array fuzzy set `res`.

`void aggregate_all();`

Aggregates all rules.

`void evaluate_all();`

Evaluates all variables.

`void defuzzy_all();`

Defuzzifies all output variables.

`void debug_rules(ostream &strm=cout, int lvl=0);`

Debugs the output on `&strm` for fuzzy rules.

12.5 CNFSet — Fuzzy set abstract base class

SYNOPSIS

```
#include <CNCL/FSet.h>
```

TYPE

`CN_FSET`

BASE CLASSES

CNNamed

DERIVED CLASSES

None

RELATED CLASSES

None

DESCRIPTION

`CNFSet` is the abstract base class for fuzzy set realizations.

Constructors:

`CNFSet(CNParam *param);`
`CNFSet(double min = 0, double max = 1);`
`CNFSet(const CNStringR xname, double min = FSET_MIN, double max = FSET_MAX);`
: Initializes a `CNFSet` with `xname` as the object's name. `min` and `max` determine the range of the membership values. `FSET_MIN` equals 0.0, `FSET_MAX` equals 1.0.

In addition to the member functions required by CNCL, `CNFSet` provides:

`virtual double get_membership(double x) const = 0;`
: Gets the membership values for `x`.

`virtual double center_of_gravity(double min, double max) const;`
: Computes the center of gravity for the defuzzyfication. (not implemented yet)

12.6 CNFSetArray — Fuzzy set based on array with membership values

SYNOPSIS

```
#include <CNCL/FSetArray.h>
```

TYPE

`CN_FSETARRAY`

BASE CLASSES

CNFSet

DERIVED CLASSES

None

RELATED CLASSES

None

DESCRIPTION

This class realizes a fuzzy set as an array. The array is containing the membership values and the index is its corrosponding x-value. Thus membership values of non-integers are interpolated.

Constructors:

`CNFSetArray();`
`CNFSetArray(CNParam *param);`
`CNFSetArray(size_t sz, double min, double max);`
Initializes `CNFSetArray`.

In addition to the member functions required by CNCL, `CNFSetArray` provides:

`virtual double get_membership(double x)const;`
Gets membership value for `x`.

`virtual double center_of_gravity(double min, double max)const;`
: Computes the center of gravity.

`double get(int i) const;`
: Gets the value from the array.

`void put(int i, double x);`
: Puts value `x` into array at position `i`.

`double &operator [] (int i);`
: Provides the direct access to the internal array.

`int get_n() const;`
: Returns the size of the array.

12.7 CNFSetLR — Fuzzy set with L and R functions

SYNOPSIS

```
#include <CNCL/FSetLR.h>
```

TYPE

`CN_FSETLR`

BASE CLASSES

CNFSet

DERIVED CLASSES

CNFSetTrapez

RELATED CLASSES

None

DESCRIPTION

`CNFSetLR` provides the LR-representation of fuzzy sets. At this representation, the interval [xm1, xm2] is considered to have the maximum membership value (1 if normalized), the left (L) and right (R) approach is described by a shape function. Additionally, a left (alpha) and a right (beta) slope is defined, thus the whole membership function is:
L ((xm1 - x) / alpha) for x <= xm1
max (1 if normalized) for xm1 <= x <= xm2
R ((x - xm2) / beta) for x >= xm2

Constructors:

```
CNFSetLR();
CNFSetLR(CNParam *param);
CNFSetLR(double xm1, double xm2, double xalpha, double xbeta, CNFuncType fL,
CNFuncType fR);
CNFSetLR(double min, double max, double xm1, double xm2, double xalpha, double xbeta,
CNFuncType fL, CNFuncType fR);
```

`CNFSetLR(CNStringR xname, double min, double max, double xm1, double xm2 double xalpha, double xbeta, CNFuncType fL, CNFuncType fR);`

`CNFSetLR(CNStringR xname, double xm1, double xm2, double xalpha, double xbeta, CNFuncType fL, CNFuncType fR);`

Initializes `CNFSetLR`. The possible variables and their default values are: left/right maximum `(xm1,xm2)` (0,0), the left/right slope `(alpha,beta)` (0,0), the left/right function `(fL,fR)` (CNFuncLin,CNFuncLin) and the values for the named `CNFSet` `xname,min,max` with the according default settings of that class.

In addition to the member functions required by CNCL, `CNFSetLR` provides:

`typedef double (*CNFuncType)(double x);`

Function pointer to the functions for L/R use. Implemented functions are:

- double CNFuncExp (double x);
  exp(-x)
- double CNFuncExp2(double x);
  exp(-x^2)
- double CNFuncLin (double x);
  1-x
- double CNFuncSqr (double x);
  1-x^2
- double CNFuncHyp (double x);
  1/(1+x)
- double CNFuncHyp2(double x);
  1/(1+x^2)

`virtual double get_membership(double x) const;`

Gets the membership values for `x`.

`double get_m1() const;`

Returns the value `m1` (left maximum).

`double get_m2() const;`

Returns the value `m2` (right maximum).

`double get_alpha() const;`

Returns the value `alpha` (left slope).

`double get_beta() const;`

Returns the value `beta` (right slope).

12.8 CNFSetTrapez — Fuzzy set with trapezium function

SYNOPSIS

`#include <CNCL/FSetTrapez.h>`

TYPE

`CN_FSETTRAPEZ`

BASE CLASSES

CNFSetLR

DERIVED CLASSES

CNFSetTriangle

RELATED CLASSES

None

DESCRIPTION

`CNFSetTrapez` realizes a trapezium shaped membership function on the base of an LR fuzzy set.

Constructors:

```
CNFSetTrapez()
CNFSetTrapez(CNParam *param);
CNFSetTrapez(double xm1, double xm2, double xalpha, double xbeta);
CNFSetTrapez(double min, double max, double xm1, double xm2, double xalpha, double
xbeta);
CNFSetTrapez(CNStringR xname, double xm1, double xm2, double xalpha, double xbeta);
CNFSetTrapez(CNStringR xname, double min, double max, double xm1, double xm2,
double xalpha, double xbeta);
```

Initialize `CNFSetTrapez` with `name` as the object's name. `xm1` is the x-value of the left maximum, `xm2` of the right one. `xalpha` is the left and `xbeta` the right slope. `min` and `max` are the function's minimum / maximum.

In addition to the member functions required by CNCL, `CNFSetTrapez` provides:

`virtual double get_membership(double x) const;`
: Returns the membership value for `x`.

12.9 CNFSetTriangle — Fuzzy set with triangle function

SYNOPSIS

```
#include <CNCL/FSetTriangle.h>
```

TYPE

`CN_FSETTRIANGLE`

BASE CLASSES

CNFSetTrapez

DERIVED CLASSES

None

RELATED CLASSES

CNFNumTriangle

DESCRIPTION

`CNFSetTriangle` realizes a triangle shaped membership function.

Constructors:

```
CNFSetTriangle();
CNFSetTriangle(CNParam *param);
CNFSetTriangle(double xm, double xalpha, double xbeta);
CNFSetTriangle(double min, double max, double xm, double xalpha, double xbeta);
CNFSetTriangle(CNStringR xname, double xm, double xalpha, double xbeta);
CNFSetTriangle(CNStringR xname, double min, double max,double xm, double xalpha,
double xbeta);
```

Initializes the `CNFSetTriangle`.

In addition to the member functions required by CNCL, `CNFSetTriangle` provides:

`virtual double get_membership(double x)const;`
: Returns the membership value for `x`.

`double get_mean() const;`
: Returns the mean value `m1==m2` (maximum).

12.10 CNFNumTriangle — Fuzzy number (triangle)

SYNOPSIS

```
#include <CNCL/FNumTriangle.h>
```

TYPE

`CN_FNUMTRIANGLE`

BASE CLASSES

CNFSetTriangle

DERIVED CLASSES

None

RELATED CLASSES

None

DESCRIPTION

`CNFNumTriangle` provides triangle fuzzy set the necessarry mathematical operations and methods for defuzzification.

Constructors:

```
CNFNumTriangle();
CNFNumTriangle(CNParam *param);
CNFNumTriangle(double m, double a, double b);
CNFNumTriangle(const CNFSetTriangle &n);
```

Initializes `CNFNumTriangle`.

In addition to the member functions required by CNCL, `CNFNumTriangle` provides:

```
double center_of_gravity(double min, double max)const;
```

Returns the center of gravity.

`double defuzzy();`
: Returns the defuzzyfied variable.

`CNFNumTriangle sqr(CNFNumTriangle);`
: Squares a `CNFNumTriangle` set, using the extension principle.

`CNFNumTriangle abs(CNFNumTriangle);`
: Returns the positiv `CNFNumTriangle`.

`CNFNumTriangle &operator =(const CNFNumTriangle &n);`
: Sets this `CNFNumTriangle` to `n`.

`CNFNumTriangle operator +(CNFNumTriangle, CNFNumTriangle);`

`CNFNumTriangle operator -(CNFNumTriangle, CNFNumTriangle);`

`CNFNumTriangle operator *(CNFNumTriangle, double);`

`CNFNumTriangle operator *(double, CNFNumTriangle);`
: Adds/ substracts two `CNFNumTriangle` sets or multiplies one with a double (extension principle).

13 Persistent Classes

The classes described here provide the possibility of persistent objects. As in most cases it is not necessary to have all the overhead for persistency inside of a running program, no complete new class need to be designed if persistency is needed. It is sufficient to add an extension to a CNCL-kompatible class where all necessary methods are included.

To generate new persistent classes or extensions, derived from any `CNCL` compatible class, the utility `CNpersistent` is provided. It generates a class framework with all functions required by CNCL and all necesarry methods for persistency.

Usage:

`CNpersistent`*[-l] [-t template-dir] basename*

The required parameter `basename` is the name of the base class. This sript will produce the persistent extension to the mentioned base class. The result are two files in the current directory: `P`*basename*`.h` (header file) and `P`*basename*`.c` (implementation file).

The optional parameters are:

- [-l] : The base class is stored in the local directory, not in the standard CNCL-tree.
- [-t template-dir] : *template-dir* is the directory where the templates will be found (if they are not in the standard directory).

Example:

```
CNpersistent MyClass
```

creates the class/extension files `PMyClass.h` and `PMyClass.c`. MyClass and CNPObjectID will be the base classes to the new extension. The only methods that have to be defined after this are the read_from constructor and the storer-method.

NOTE:

- At some classes it will be necessary to achieve direct access to some private members of the base class, e.g. for the read_from constructor or the storer-method. In order to achieve this it is important to declare the persistent class or extension as *friend class* at the base class.
- All extensions have the complete *Default I/O member function for CNCL classes*. In most cases this I/O is not necessary because all these functions are already defined in the base class. Thus all of them can be deleted (they are marked with the note *(to be deleted if not needed))*.
- Please don't wonder that the rule of no multiple inheritance inside of CNCL seems to be broken at the persistent classes. The second base class, `CNPObjectID`, is not derived from any other CNCL-class. So the problems of multiple inheritance can not occur.

13.1 CNReaderTbl – Table for adress of reader-function

SYNOPSIS

`#include <CNCL/ReaderTbl.h>`

TYPE

`CN_READERTBL`

BASE CLASSES

CNCL

DERIVED CLASSES

None

RELATED CLASSES

None

DESCRIPTION

`CNReaderTbl` provides a table of all persistent classes (`Reader-Table`) and a second table of id's `ID-Table`. Both tables are realized as dynamic hash-tables.

The first one uses the classname as key for registering in the table, the object stored in it is the reader function pointer (the adress of the reader-function). This table is for checking if a class is registered to be persistent and for calling the reader constructor when objects are taken from the stream.

The other table uses the id as key and stores the according object to this id. The table is used during two occasions. First whenever an object is written to the stream the id is stored in this table. When the same object should be stored again it will be recognized. In addition to the id a special delimitation character is used for indicating this. When the objects are taken from the stream all objects and their old id are stored in this table again. When this special delimitation character apears on the stream the object can be taken out of the table by its id. Thus it is possible to restore containers even if one object appears several times at different locations, e.g at a double linked list.

Constructors:

`CNReaderTbl(char* classname, Reader_ptr reader_ptr);`
: Initializes a new Reader Table entry. Each given `classname` is added to the table.

Additional types provided by `CNReaderTbl` are:

`typedef CNObject*(*Reader_ptr)(CNPstream&);`
: This is the type for the reader-function-pointer.

The functions provided by `CNReaderTbl` are:

`static Reader_ptr reader_pointer_by_classname(char* classname);`
: This function checks if the requested `classname` is registered in the Reader Table and returns the according Reader_ptr. If it is not registered, a fatal error is yielded.

`static bool is_in(long id_key);`
: Checks if the given `id_key` is in the id table.

`static void add_to_id_tbl(long id_key, CNObject* obj);`

`static void add_to_id_tbl(long id_key, CNObject& obj);`
: Adds the persistent object `obj` to the id table with `id_key` as its key.

`static CNObject* get_from_id_tbl(long id_key);`
: Returns the persistent object stored in the id table with `id_key` as the key.

`static void reset_id_tbl();`
: Resets the id table to its initial state (all stored elemts are deleted). This method should be used if You change from the `write-to-stream` to the `read_from_stream` sequence.

13.2 PObjectID — ID-Managment for persistent Objects

SYNOPSIS

`#include <CNCL/PObjectID.h>`

TYPE

BASE CLASSES

None

DERIVED CLASSES

all persistent classes and extensions

RELATED CLASSES

None

DESCRIPTION

This class manages the (unique) identification number of each persistent object. All persistent classes must be derived from this class. As `CNPObjectID` is not derived from any CNCL-class `multiple inheritance` can be used.

One of the member variables, declared as static, is the counter for the actual id. It is increased whenever a new id is given to an object. The other one is the id of the object itself. This number cannot occur two times inside of one program.

Constructors:

`CNPObjectID();`
: Initializes the id and increases the `static` counter. This constructor must be called in all constructors of the derived classes.

Additional types provided by `CNReaderTbl` are:

`typedef long PID`
For a possible future change from long to any other larger number this typedef has been included.

`CNPObjectID` provides the following functions:

`CNPID object_id();`
Returns the object id.

13.3 CNPIO — persistent stream Object IO-formatting

SYNOPSIS

`#include <CNCL/PIO.h>`

TYPE

`CN_PIO`

BASE CLASSES

CNCL

DERIVED CLASSES

None

RELATED CLASSES

None

DESCRIPTION

`CNPIO` provides the format of storing and reading the persistent objects on/from a stream.The format convention can be described as follows:
`classname`
`delimitation character`
`data`
`delimitation character`

The functions provided by `CNPIO` are:

`static int store_object(CNPstream& stream, CNObject& obj, bool no_ptr_check = FALSE);`
`static CNObject* read_object(CNPstream& stream);`
: Reads/Stores a *persistent* `CNObject` on the `stream`. A `CNObject` is recognized to be persistent if it is registered at the actual `Reader Table`. `no_ptr_check` determines if the objects are checked for multiple appearence (`no_ptr_check = FALSE`) or not (`TRUE`).

: Stores the id of the persistent object `obj` to the given stream.

`static CNPID read_id(CNPstream& stream);`
: Reads the id from a stream.

13.4 CNPstream — abstract base class for persistent stream classes

SYNOPSIS

```
#include <CNCL/Pstream.h>
```

TYPE

CN_PSTREAM

BASE CLASSES

CNObject

DERIVED CLASSES

None

RELATED CLASSES

None

DESCRIPTION

`CNPstream` is the abstract base class for the persistent stream classes. Thus, only IO-operators and IO-functions are defined here. The concrete definition of the methods and operators must be done in the derived classes. Thus by this approach the possibilities of polymorphism are kept.

Constructors:

```
CNPstream();
CNPstream(*param);
```

Initializes the `CNPstream`.

In addition to the member functions required by CNCL, `CNstream` provides:

```
virtual CNPstream& operator<<(const char*)=0;
virtual CNPstream& operator<<(char)=0;
```

`virtual CNPstream& operator<<(long)=0;`

`virtual CNPstream& operator<<(double)=0;`

`virtual CNPstream& operator>>(char*)=0;`

`virtual CNPstream& operator>>(char&)=0;`

`virtual CNPstream& operator>>(long&)=0;`

`virtual CNPstream& operator>>(double&)=0;`
: Virtual persistent IO operators.

`virtual CNPstream& getline(char*, int ) = 0;`
: Virtual `getline` function for string input.

13.5 CNPiostream — persistent iostream format

SYNOPSIS

`#include <CNCL/Piostream.h>`

TYPE

`CN_PIOSTREAM`

BASE CLASSES

CNPstream

DERIVED CLASSES

None

RELATED CLASSES

None

DESCRIPTION

The class `CNPiostream` manages the interface to the gnu `c++` iostream library. Thus, all virtual functions of the base class `CNPstream` are defined here. For a description see the base class.

Constructors:

`CNPiostream(iostream& s);`
`CNPiostream(CNParam *param);`
Initializes the `CNPiostream`.

13.6 CNPInt — class persistent CNInt

SYNOPSIS

```
#include <CNCL/PInt.h>
```

TYPE

`CN_PINT`

BASE CLASSES

CNInt, CNPObjectID

DERIVED CLASSES

None

RELATED CLASSES

CNPString, CNPDouble

DESCRIPTION

`CNPInt` manages the persistency of `CNInt`'s. As `CNInt` is its base class, all methods of `CNInt` are available.

Constructors:

```
CNPInt(long val=0);
CNPInt(CNPstream& stream);
CNPInt(CNParam *param);
```

Initializes the `CNPInt`. Either the value is given as `val` (0 by default) or it will be read from the persistent `stream` (reader-constructor).

In addition to the member functions required by CNCL, `CNInt` provides:

```
virtual int store_on(CNPstream& s);
```

`int store_on(CNPstream& s, bool no_ptr_check);`
: Stores the `CNPInt` on the persistent stream `s`. The boolean parameter `no_ptr_check` switches the check for multiple storing off (if set *TRUE*).

`static CNPInt* read_from(CNPstream& s);`
: Reads the `CNPInt` from stream `s`.

`static CNObject* object_read_from(CNPstream& s);`
: Reads from stream `s` as a `CNObject`.

`CNPID object_id();`
: Returns the ID of the current Object.

`virtual int storer(CNPstream&);`

`static CNPInt* reader(CNPstream& s);`
: These two functions are called by the class `CNPIO` as a connection to the persistent output/reader-construktor.

13.7 CNP<type> — persistent types

DESCRIPTION

CNCL currently provides persistent extensions for the (CNCL-) data types `CNDouble`, `CNInt`, `CNString` and the container classes `CNDLList`, `CNArrayObject`.

The only difference of functions in these extensions to the previosly described extension `CNPInt` may be some additional constructors and =operators for a better connection to the base class.

Concept Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

Function Index

~

<

A

B

C

D

E

F

G

P

R

S

Z

Table of Contents