ObjectDataBase

Release 0.1.17

Jul 29, 2018

Contents:

CHAPTER 1

COdb

The OjectDatabase is a Graph-Database.

see also: https://en.wikipedia.org/wiki/Graph_database

It aids you in your endeavor to warp the world around you. A GraphDB has no actual data structure, it has a data management structure but without any force to drive you in a specific direction.

So you may build your own world in your DB as you like it. Learning in depth about its nature.

odb was testet with inhaling the whole IMDB with about 30Mio orignal data records using only 27GB of RAM. If you wish to test it for yourself (having 30GB free memory), build the odb an do th following:

```
cd ../raw-data
cat imdb.calls
# do all curl calls
gunzip *.gz
cd ../build
./7-read-imdb.test 20000000
```

It yields a statistic like this:

```
---------------- 12mio things
---------------- 15mio properties
---------------- 5mio reasons
---------------- 0 atoms

---------------- Search in (t)hings or ... or (s)ave&quit:
```

Now you can search the IMDB using a complete new perspective:

```
---------------- Search in (t)hings or (r)easons or (p)roperties ...: p

---------------- Search in Properties: .*:HU:.*

1:Carmencita - spanyol tánc:HU::imdbDisplay::0:
```

(continues on next page)

(continued from previous page)

```
  Property of: Carmencita

2:A bohóc és kutyái:HU::imdbDisplay::0:
  Property of: Le clown et ses chiens

---------------- Search in (t)hings or (r)easons or (p)roperties ...: t

---------------- Search in Things: .*Star.*

Demolishing and Building Up the Star Theatre
  Property: 1901
  Property: 1:Building Up and Demolishing the Star Theatre:US::dvd::0:
  Property: 2:    :RU::::0:
  Property: 3:Demolishing and Building Up the Star Theatre:::original::1:
  Property: 4:Star Theatre:US:::copyright title:0:
  Property: 5:A Star Theatre lebontása és felépítése:HU::imdbDisplay::0:
  Property: 6:Demolishing and Building Up the Star Theatre:US::::0:
  Property: Documentary
  Property: Short
  Property: class:movie
  Property: short
  Property: tt0000410
```

odb may support you to build a natural image of relevant links between CThing's amongst each other using named, unidirectional CReason's as well as appending things like CProperty's (descriptive mini data elements), CAtom's (true data fields) to CThing's.

You may use the odb linked in your development environment.

Sample code

```
#include <iostream>

#include "odb.h"

int main()
   {
   auto oOdb = odb::COdb();
   oOdb.print_json( std::cout );
   std::cout << '\n';
   }
```

class COdb : **public** Identifiable<*COdb*>

The database.

A Object Database itself is an Identifiable object. Enabeling multiple database instances in one application.

Author Manfred Morgner

Since 0.1.17

Public Functions

COdb()

Creates a *COdb*, a database.

We only default construct the database, no copy construction no copy of the database at all. It is not known, what copying the database means.

COdb(*COdb* **const** &*src*)
deleted: copy constructor

Copying a databse is not supported yet. We will support it as soon as we find out what it means.

COdb(*COdb* **const** &&*src*)
deleted: move constructor

Moving a databse is not supported yet.

COdb &**operator=**(*COdb* **const** &*src*)
deleted: Assignment operator

Here we are in the same situation as with copy construction

virtual ~COdb()
Some cleanup.

If we wish to convience valgrind we are doing it right, we have to free all links to other objects before leaving the show.

void **clear**()
Frees all objects.

Frees all known objects at last so far that freeing the object collections releases all memory, bound to them. So valgrind will find nonode left on the table.

PNode **MakeNode**(std::string **const** &*crsName* = "")
Creates a PNode.

Creates a shared_ptr with a new *CNode* named as given in the call. If no name is given, the name will be the class default

Parameters

- `crsName`: The name for the *CNode*

PNode **LoadNode**(size_t *nId*, std::string **const** &*crsName* = "")
Creates a PNode with predefined ID.

Creates a shared_ptr with a new *CNode* named as given in the call. If no name is given, the name will be the class default

Parameters

- `nId`: The predefined ID if loading given sets into odb
- `crsName`: The name for the *CNode*

PProperty **MakeProperty**(std::string **const** &*crsName* = "")
Creates a PProperty.

Creates a shared_ptr with a new *CProperty* named as given in the call. If no name is given, the name will be the class default

Parameters

- `crsName`: The name for the *CProperty*

PProperty **LoadProperty**(size_t *nId*, std::string **const** &*crsName* = "")
Creates a PProperty with predefined ID.

Creates a shared_ptr with a new *CProperty* named as given in the call. If no name is given, the name will be the class default

Parameters

- `nId`: The predefined ID if loading given sets into odb
- `crsName`: The name for the *CProperty*

PAtom **MakeAtom**(int *data*, std::string **const** &*crsName* = "", std::string **const** &*crsPrefix* = "", std::string **const** &*crsSuffix* = "", std::string **const** &*crsFormat* = "")
Creates a PAtom.

Creates a shared_ptr with a new *CAtom* named as given in the call. If no name is given, the name will be the class default

The data type of the data element in *CAtom* follows the input data type. It can be if any primitive type or most of the simple containers, like string or vector.

CAtom further on manages the life time of the data element. It's a unique_ptr

Parameters

- `data`: The data for the *CAtom*
- `crsName`: The name for the *CAtom*
- `crsPrefix`: The prefix for the *CAtom* in a GUI
- `crsSuffix`: The suffix for the *CAtom* in a GUI
- `crsFormat`: The format for the *CAtom* in a GUI

PAtom **LoadAtom**(size_t *nId*, int *data*, std::string **const** &*crsName* = "", std::string **const** &*crsPrefix* = "", std::string **const** &*crsSuffix* = "", std::string **const** &*crsFormat* = "")
Creates a PAtom with predefined ID.

Creates a shared_ptr with a new *CAtom* named as given in the call. If no name is given, the name will be the class default

The data type of the data element in *CAtom* follows the input data type. It can be if any primitive type or most of the simple containers, like string or vector.

CAtom further on manages the life time of the data element. It's a unique_ptr

Parameters

- `nId`: The predefined ID if loading given sets into odb
- `data`: The data for the *CAtom*
- `crsName`: The name for the *CAtom*
- `crsPrefix`: The prefix for the *CAtom* in a GUI
- `crsSuffix`: The suffix for the *CAtom* in a GUI
- `crsFormat`: The format for the *CAtom* in a GUI

PReason **MakeReason**(std::string **const** &*crsName* = "")
Creates a PReason.

Creates a shared_ptr with a new *CReason* named as given in the call. If no name is given, the name will be the class default

Parameters

- `crsName`: The name for the *CReason*

PReason **LoadReason**(size_t *nId*, std::string **const** &*crsName* = "")
Creates a PReason with predefined ID.

Creates a shared_ptr with a new *CReason* named as given in the call. If no name is given, the name will be the class default

Parameters

- `nId`: The predefined ID if loading given sets into odb
- `crsName`: The name for the *CReason*

PStrand **MakeStrand**(std::string **const** &*crsName* = "")
Creates a PStrand.

Creates a shared_ptr with a new *CStrand* named as given in the call. If no name is given, the name will be the class default

Parameters

- `crsName`: The name for the *CStrand*

void **print**() **const**
Print out the database (Informative format)

void **print**(CAtoms **const** &*crContainer*) **const**
Print out container of *CAtom* objects.

Parameters

- `crContainer`: The forward iterable container, containing PAtom's

template <**typename** T>
void **print**(CT<T> **const** &*crContainer*) **const**
Print out container of *CNode*'s.

Parameters

- `crContainer`: The forward iterable container, containing all *CNode* instances

auto **Escape**(std::string **const** &*crsInput*)
Replaces 2 with " and \ with .

Parameters

- `crsInput`: The string to be escaped

void **print_json**(CNodes **const** &*crContainer*, std::ostream &*ros*)
Dump all CNodes in Sub-JSON format.

Parameters

- `crContainer`: The forward iterable container, containing all *CNode* instances
- `ros`: The output destination

void **print_json_stream**(CNodes **const** &*crContainer*, std::ostream &*ros*)
Dump all CNodes in Sub-JSON format.

Parameters

- `crContainer`: The forward iterable container, containing all *CNode* instances
- `ros`: The output destination

void **print_json**(CProperties **const** &*crContainer*, std::ostream &*ros*)
Dump all *CProperty*'s in Sub-JSON format.

Parameters

- `crContainer`: The forward iterable container, containing all *CProperty* instances
- `ros`: The output destination

void **print_json_stream**(CProperties **const** &*crContainer*, std::ostream &*ros*)
Dump all *CProperty*'s in Sub-JSON format.

Parameters

- `crContainer`: The forward iterable container, containing all *CProperty* instances
- `ros`: The output destination

void **print_json**(CAtoms **const** &*crContainer*, std::ostream &*ros*)
Dump all CAtoms in Sub-JSON format.

Parameters

- `crContainer`: The forward iterable container, containing all *CAtom* instances
- `ros`: The output destination

void **print_json_stream**(CAtoms **const** &*crContainer*, std::ostream &*ros*)
Dump all CAtoms in Sub-JSON format.

Parameters

- `crContainer`: The forward iterable container, containing all *CAtom* instances
- `ros`: The output destination

void **print_json**(CReasons **const** &*crContainer*, std::ostream &*ros*)
Dump all CReasons in Sub-JSON format.

Parameters

- `crContainer`: The forward iterable container, containing all *CReason* instances
- `ros`: The output destination

void **`print_json_stream`**(CReasons **`const`** &*crContainer*, std::ostream &*ros*)
Dump all CReasons in Sub-JSON format.

Parameters

- `crContainer`: The forward iterable container, containing all *CReason* instances
- `ros`: The output destination

void **`print_json`**(std::ostream &*ros*)
Dump the hole database in JSON format.

see also: JSON validator+converter: https://jsonformatter.org/

Sample: Link Atoms to Nodes and Node to Node and dump it as JSON

```
#include <iostream>

#include "odb.h"
#include "atom.h"
#include "node.h"

int main()
    {
    auto oOdb    = odb::COdb();
    auto pNode1 = oOdb.MakeNode("Ulrich");
    auto pNode2 = oOdb.MakeNode("Fred");
    auto pAtom1  = oOdb.MakeAtom("Leader", "Role");
    auto pAtom2  = oOdb.MakeAtom("Member", "Role");
    auto pReason = oOdb.MakeReason("pays");
    pNode1->Append(pAtom1);
    pNode2->Append(pAtom2);
    pNode1->Link(pNode2, pReason);
    oOdb.print_json(std::cout);
    }
```

Output

```
{
"Object Database Dump":
    {
    "Sizes": [ {"P": 0},{"A": 2},{"R": 1},{"T": 2} ],
    "Properties":
        [
        ],
    "Atoms":
        [
            { "id": 0, "data": "Leader" },
            { "id": 1, "data": "Member" }
        ],
    "Reasons":
        [
            { "id": 0, "name": "pays" }
        ],
```

(continues on next page)

(continued from previous page)

```
    "Nodes":
        [
            { "id": 0, "name": "Ulrich",
                "properties": [  ],
                "atoms": [ {"id": 0} ],
                "links": [ {"node-id": 1, "reason-id": 0} ] },
            { "id": 1, "name": "Fred",
                "properties": [  ],
                "atoms": [ {"id": 1} ],
                "links": [  ] }
        ]
    }
}
```

Parameters

- `ros`: The output destination

void **`print_json_stream`**(std::ostream &*ros*)
Dump the hole database in JSON format.

Parameters

- `ros`: The output destination

void **`SaveDB`**(std::string **`const`** &*crsFilename*)
Saves an odb json file.

Parameters

- `crsFilename`:

bool **`LoadDB`**(std::string **`const`** &*crsFilename*)
Loads an odb json file.

Parameters

- `crsFilename`:

PNode **`FindOrLoadNodeById`**(size_t **`const`** *nId*, std::string **`const`** &*crsName* = "")
Has to return a node with specified ID, if it does not exists, it is to make.

Parameters

- `nId`: The id of the node
- `crsName`: The name of the node if it has to be created

ONode **`FindNodeByProperty`**(std::string **`const`** &*crsProperty*)
Finds PNode with a named Property only if it's unique.

Parameters

- `crsProperty`: The name for the *CProperty*

CNodes **FindNodesByProperty**(std::string **const** &*crsProperty*)
Finds PNode with a named Property only if it's one.

Parameters

- `crsProperty`: The name for the *CProperty*

CNodes **FindNodesByProperty**(std::regex **const** &*crsRegex*)
Finds PNode with a named Property only if it's one.

Parameters

- `crsRegex`: The name for the *CProperty*

PNode **FindOrMakeNodeByProperty**(std::string **const** &*crsNode*, std::string **const** &*crsProperty*)
Finds or creates a PNode with a named Property, which also may be created and assigned.

Parameters

- `crsNode`: The name for the *CNode*
- `crsProperty`: The name for the *CProperty*

MLinkets **FindNodeLinkingSameNode**(size_t **const** *cnIdNodeA*, size_t **const** *cnIdNodeB*)
Returns the nodes linked by the same reason as the given Nodes.

Parameters

- `cnIdNodeA`: Id of one Node
- `cnIdNodeB`: Id of the other one

PNode **FindOrMakeNode**(std::string **const** &*crsNode*)
Has to return a node, if it does not exists, it is to make.

Parameters

- `crsNode`: The name of the Node

PProperty **FindOrMakeProperty**(std::string **const** &*crsProperty*)
Has to return a property, if it does not exists, it is to make.

Parameters

- `crsProperty`: The name of the Property

PReason **FindOrMakeReason**(std::string **const** &*crsReason*)
Has to return a Reason, if it does not exists, it is to make.

Parameters

- `crsReason`: The name of the Reason

PAtom **FindOrMakeAtom**(std::string **const** &*crsAtom*)
Has to return an Atom, if it does not exists, it is to make.

Parameters

- `crsAtom`: The name of the Atom

bool **AppendProperty2Node**(size_t *nProperty*, size_t *nNode*)
todo: optimize / Appends a Property to a Node by given index value

bool **AppendProperty2Node**(std::string **const** &*crsProperty*, std::string **const** &*crsNode*)
todo: optimize / Appends a Property to a Node by given names

bool **AppendProperty2Node**(std::string **const** &*crsProperty*, bool **const** *bForce*, std::string **const** &*crsNode*)
todo: optimize / Appends a Property to a Node by given names

bool **AppendAtom2Node**(size_t *nNode*, size_t *nAtom*)
todo: optimize / Appends an Atom to a Node by given index value

bool **LinkNode2Node**(size_t *nNodeFrom*, size_t *nReason*, size_t *nNodeTo*)
todo: optimize / Links a Node to a Node for a Reason by given index value

bool **LinkNode2Node**(std::string **const** &*crsNodeFrom*, std::string **const** &*crsReason*, std::string **const** &*crsNodeTo*)
todo: optimize / Links a Node to a Node for a Reason by given names

template <typename T>
std::optional<PT<T>> **Find**(CT<T> **const** &*croContainer*, size_t *nId*)
Finds the T with ID nId.

Template Parameters

- `T`: the type of the filtered objects

Parameters

- `croContainer`: The container to be filtered
- `nId`: The ID of the T

auto **FindNode**(size_t *nId*)
API Adapter.

auto **FindNodes**(std::string **const** &*crsName*)
API Adapter.

auto **FindNodes**(std::regex **const** &*crsRegex*)
API Adapter.

auto **FindProperty**(size_t *nId*)
API Adapter.

auto **FindProperties**(std::string **const** &*crsName*)
API Adapter.

auto **FindProperties**(std::regex **const** &*crsRegex*)
API Adapter.

auto **FindReason**(size_t *nId*)
API Adapter.

auto **FindReasons**(std::string **const** &*crsName*)
API Adapter.

auto **FindReasons**(std::regex **const** &*crsRegex*)
API Adapter.

auto **FindAtom**(size_t *nId*)
API Adapter.

auto **FindAtoms**(std::string **const** &*crsName*)
API Adapter.

auto **FindAtoms**(std::regex **const** &*crsRegex*)
API Adapter.

auto **FindUnUsedNodes**()
find all nodes not linked with anynode

auto **FindUnUsedReasons**()
find all reasons not used with anynode

auto **FindUnUsedProperties**()
find all properties not linked to anynode

auto **FindUnUsedAtoms**()
find all atoms not linked to anynode

auto **const** &**Nodes**() **const**
Access function to call then container of *CNode*'s.

auto **const** &**Properties**() **const**
Access function to call then container of CProperties.

auto **const** &**Atoms**() **const**
Access function to call then container of *CAtom*'s.

auto **const** &**Reasons**() **const**
Access function to call then container of *CReason*'s.

auto **const** &**Strands**() **const**
Access function to call then container of *CStrand*'s.

CHAPTER 2

CNode

The Node, formerly known as Object in the Object Database, is an entity representing some 'thing' or some 'person'. Or, if you see animals neigther as thing nor as person, some 'animal'. We are living in a database world so this distinguation makes no difference, at least for the database.

A node may contain an arbitrary amount of arbitrary

- CAtom's (equivalent to Data Fields)
- CProperty's (minimal data units, multiusable)
- Links to other CNode's for specified CReason's
- as well as Backlinks to Reason'ed Link sources.

Basically the linking CNode manages the following resources:

- from node to node
- the backlink for links from itself to another nodes
- from node to atoms
- the backlink for links from itself to atoms

Sample code

```
/**
    @file nodes.cpp

    @author Manfred Morgner
    @date 15.04.2018

    Demonstration of how Nodes are made and how they can be connected to
    each other using specific unidirectional reasons.
 */

#include <iostream>
```

(continues on next page)

(continued from previous page)

```
#include "odb.h"

auto oOdb = odb::COdb();


// Demo main program
int main()
    {
    // 3 people
    oOdb.MakeNode("Udo");
    oOdb.MakeNode("Ina");
    oOdb.MakeNode("Rob");

    // 3 kind of relation
    oOdb.MakeReason("is father of");
    oOdb.MakeReason("knows");
    oOdb.MakeReason("loves");

    // 3 bindings
    oOdb.LinkNode2Node("Udo", "is father of", "Ina");
    oOdb.LinkNode2Node("Udo", "knows"       , "Rob");
    oOdb.LinkNode2Node("Ina", "loves"       , "Rob");

    // show us
    std::cout << "---------------- all nodes" << '\n';
    for ( auto const & a:oOdb.Nodes() )
        {
        std::cout << *a << '\n';
        }
    }
```

Output

```
---------------- all nodes
Udo
   => linked to: "Ina" for reason: "is father of"
   => linked to: "Rob" for reason: "knows"
Ina
   => linked to: "Rob" for reason: "loves"
   <= linked from: Udo
Rob
   <= linked from: Ina
   <= linked from: Udo
```

class CNode : **public** enable_shared_from_this<*CNode*>, **public** Identifiable<*CNode*>

A Node as you imagine it.

A *CNode*, or Node, is a main structure element in a GrapDB like odb. The counterpart is a *CReason*. *CNode*'s will be linked unidirectional to another *CNode*'s. There may be an arbitrary amount of Links between CNodes e.g.

- Node, (Link-)Reason, Node
- Mary, wrote, a book
- Mary, read, a book
- Mary, loves, Jack

Author Manfred Morgner

Since 0.1.17

Public Functions

CNode()
DELETED: We never construct without a name for the node.

CNode(*CNode* **const**&)
DELETED: and we don't make copies.

CNode(*CNode*&&)
Move-Contructor, noexcept and default. make_shared<T> move-constructs. Function return of *CNode*'s moves too.

CNode(std::string **const** &*crsName*)
Normal constructor, receiving the name of the Node.

CNode(size_t *nId*, std::string **const** &*crsName*)
Load constructor, receiving the ID and name of the Node.

virtual ~CNode()
Destructor (default). Nothings special here.

void **clear**()
We need to unbind all relations in the *CObj* before letting us destruct

PProperty **Append**(PProperty *poProperty*)
Appends an *CProperty* to its property list.

Appending an *CProperty* to this *CNode* includes the Node to inform the appended Property about this Node is linking to it

Parameters

- `poProperty`: A Property to bind with the Node

PAtom **Append**(PAtom *poAtom*)
Appends an *CAtom* to its atom list.

Appending an *CAtom* to this *CNode* includes the Node to inform the appended Atom about this *CNode* is linking to it

Parameters

- `poAtom`: An Atom to bind into the Node

PNode **Link**(PNode *po2Node*, PReason *po4Reason*)
Links this *CNode* to another *CNode* for a *CReason*.

Parameters

- `po2Node`: A Node to Link to
- `po4Reason`: The Reason we link for

PNode **Unlink**(PNode *po2Node*, PReason *po4Reason*)
Removes a link to a specific *CNode* with a specific *CReason*.

Parameters

- `po2Node`: A Node to Linked to
- `po4Reason`: The Reason we linked for

PNode **RelatingNodeAdd**(PNode *poNode*)
adds a *CNode* as referencing to this

Parameters

- `poNode`: A *CNode* that links to us notifies us, we register it

PNode **RelatingNodeSub**(PNode *poNode*)
subtract a *CNode* as referencing to this

Parameters

- `poNode`: A *CNode* that linked to us notifies us, we deregister it

auto **IsUnUsed**()
returns if the instance is 'free'

MLinkets **const** &**Linkets**() **const**
Nodes linking to this Node.

SNodes **const** &**Nodes**() **const**
Nodes linked by this Node.

SProperties **const** &**Properties**() **const**
Properties assigned to this Node.

SAtoms **const** &**Atoms**() **const**
CAtoms used by this Node.

Public Static Attributes

constexpr auto **s_csNameUnnamedNode** = {"unnamedNode"}
The name of the node.

constexpr bool **s_bDebug** = {false}
Do we generate debug output?

Friends

std::ostream &**operator<<**(std::ostream &*ros*, *CNode* **const** &*crNode*)
The free output operator for *CNode*.

The ouput operator prints all information about the *CNode* instance to the given output stream. This is:

- Name
- *CProperty*'s

- *CAtom*'s
- Link destinations + *CReason*
- Backlinks

Parameters

- `ros`: The output stream to send the Node to
- `crNode`: The Node to output

CHAPTER 3

CProperty

Sample code

```
/**
 *  @file main.cpp
 *
 *  @author Manfred Morgner
 *  @date 10.02.2018
 */

#include <iostream>

#include "odb.h"


/// @brief Demo main program for "property in node"
int main()
   {
   auto oOdb = odb::COdb();
   auto node = oOdb.MakeNode( "Tree" );
   auto property = oOdb.MakeProperty( "Acorn" );
   node->Append(property);
   std::cout << "node: " << *node;
   std::cout << '\n';
   }
```

Output

```
thing: Tree
  Property: Acorn
```

class CProperty : **public** Identifiable<*CProperty*>

A Property for a *CNode*.

Public Functions

CProperty()
DELETED: default constructor.

CProperty(*CProperty* **const**&)
DELETED: There is no reason to copy a *CProperty*.

CProperty(*CProperty*&&)
DEFAULT, NOEXCEP: There is no reason to move-construct a *CProperty*.

CProperty(std::string **const** &*crsName*)
Normal constructor, receiving the name of the property.

CProperty(size_t *nId*, std::string **const** &*crsName*)
Load constructor, receiving the id and the name of the property.

virtual ~CProperty()
DEFAULT, NOEXCEPT: destructor.

operator std::string const&()
Conversion operator will return the name of the instance.

void **RelationAdd**(PNode *poNode*)
Add the information about a link from a *CNode*.

void **RelationSub**(PNode *poNode*)
Removes a link to a *CNode*.

Parameters

- `poNode`: The node the link is pointing to

void **print**()
Prints an informational output to std::cout.

auto **IsUnUsed**()
returns if the instance is 'free'

SNodes **const** &**Relations**() **const**
Access function to call then container of PNodes's.

Public Static Attributes

constexpr auto **s_csNameUnnamedProperty** = {"unnamedProperty"}
The name of an unnamed property.

Friends

std::ostream &**operator<<**(std::ostream &*ros*, *CProperty* **const** &*croProperty*)
Output operator to do an output of the instance.

CHAPTER 4

CReason

CReason is necessary to link two CThing instances to give the link an explaination. This enables to link the same CThing's multiple times.

```
thing1 - reason - thing2
```

For example:

```
Heinz - wrote - 'Trees of Estonia'
Heinz - signed - 'Trees of Estonia'
```

Links are unidirectional. Meaning if it's true that

```
Heinz - wrote - 'Trees of Estonia'
```

it may not be true that

```
'Trees of Estonia' - wrote - Heinz
```

To ensure thing2 feels the link, it will be informed that a link to it became established. The linked thing registers which thing is linking to it only ones. In our example 'Trees of Estonia' registers, that Heinz links to it.

If some process/entity needs to know how often and for which reasons, it has to go to Heinz and ask. The linking thing cares about correct management of links, reasons and backlinks.

CReason registers each link it is used for

Demostration

```
#include <iostream>

#include "odb.h"
#include "thing.h"

int main()
    {
```

(continues on next page)

(continued from previous page)

```
    // generating the objects
    auto oOdb    = odb::COdb();
    auto pThing1 = oOdb.MakeThing("Ulrich");
    auto pThing2 = oOdb.MakeThing("Fred");
    auto pReason = oOdb.MakeReason("pays");
    // create a connection
    pThing1->Link(pThing2, pReason);
    // let them explain the situation
    std::cout << "thing: " << *pThing1 << '\n';
    std::cout << "thing: " << *pThing2 << '\n';
    }
```

Output:

```
thing: Ulrich
   => linked to: "Fred" for reason: "pays" = Ulrich pays Fred
thing: Fred
   <= linked from: Ulrich
```

class CReason : **public** Identifiable<*CReason*>

A Reason to link two Nodes (Unidirectional)

Public Functions

CReason()

DELETED: Default constructor.

CReason(*CReason* **const**&)

DELETED: There is no reason to copy a *CReason*.

CReason(*CReason*&&)

DEFAULT, NOEXCEPT: move constructor, make_shared<T> move-constructs returning objects has to move too, we don't want copies!

CReason(std::string **const** &*crsName*)

Normal constructor, receiving the name of the reason.

CReason(size_t *nId*, std::string **const** &*crsName*)

Load constructor, receiving the ID and name of the reason.

virtual ~CReason()

DEFAULT, NOEXCEPT: destructor.

operator std::string const&()

Conversion operator will return the name of the instance.

void **RelationAdd**(PNode &*poNodeFrom*, PNode &*poNodeTo*)

Add the information about linking from one *CNode* to another regarding 'this' *CReason*

void **RelationSub**(PNode &*poNodeFrom*, PNode &*poNodeTo*)

Removes a link between two *CNode*'s.

Removes the information about a particular link from one *CNode* to another regarding 'this' reason

Parameters

- `poNodeFrom`: The node the link is claimed to be made from
- `poNodeTo`: The node the link is claimed to be made to

void **`print`**`()`
Prints an informational output to std::cout.

auto **`IsUnUsed`**`()`
returns if the instance is 'free'

MLinks **`const`** &**`Relations`**`()` **`const`**
Returns the links this Reason administers.

Public Static Attributes

`constexpr` auto **`s_csNameUnnamedReason`** = {"unnamedReason"}
The name of an unnamed reason.

Friends

std::ostream &**`operator<<`**(std::ostream &*ros*, *CReason* **`const`** &*croReason*)
Output operator to do an output of the instance.

CHAPTER 5

CAtom

An CAtom is a container for a single data element, let's say a number or a text. It stores additional information to use in a GUI as there are

- (Field-)Name
- Prefix
- Suffix
- (output) Format template

One can compare an Atom with single data field as in a conventional table oiented database. Unlike conventional databases fields/atoms do not have a fixed structure, they even do not have to exist.

CAtom may act as a template for other atoms. In such case the atom, using the other as template, does not need to fill elements which are given by the template. It will appear as if the elements of the template are elements of the using Atom, as long as they are not overwritten.

Sample code

```
/**
    @file atom-test.cpp

    @author Manfred Morgner
    @date 21.01.2018
 */

#include <iostream>

#include "odb.h"


/**
    @brief Demo main program
 */
int main()
```

(continues on next page)

(continued from previous page)

```
    {
    auto oOdb = odb::COdb();
    auto atom = oOdb.MakeAtom(2.5, "gain", "is", "%");
    std::cout << "atom data: " << *atom << '\n';
    std::cout << "atom frmt: " << atom->m_sName << ' ';
    atom->print_atom_data_formated(std::cout);
    std::cout << '\n';
    oOdb.print_json(std::cout);
    } // int main()
```

Output

```
atom data: 2.5
atom frmt: gain is 2.5 %
```

class CAtom : **public** enable_shared_from_this<*CAtom*>, **public** Identifiable<*CAtom*>

An Atom is a data field for a *CNode*.

Sample Code goes here

```
#include <odb>
```

Public Functions

template <typename T>
CAtom(T *tAtomData*, std::string **const** &*crsName* = "", std::string **const** &*crsPrefix* = "", std::string **const** &*crsSuffix* = "", std::string **const** &*crsFormat* = "")

Constructor able to receive data of maany types.

Parameters

- `tAtomData`: The data unit to encapsulate
- `crsName`: The name for the atom
- `crsPrefix`: The prefix for user output
- `crsSuffix`: The suffix for user output
- `crsFormat`: The format for user output

template <typename T>
CAtom(size_t *nId*, T *tAtomData*, std::string **const** &*crsName* = "", std::string **const** &*crsPrefix* = "", std::string **const** &*crsSuffix* = "", std::string **const** &*crsFormat* = "")

Constructor able to receive data of maany types.

Parameters

- `nId`: The predefiined ID if loading a dataset
- `tAtomData`: The data unit to encapsulate
- `crsName`: The name for the atom
- `crsPrefix`: The prefix for user output
- `crsSuffix`: The suffix for user output

- `crsFormat`: The format for user output

virtual ~CAtom()
Destruction as usual (=default)

void **clear**()
Remove all links between all objects.

This is necessary to enable freeing of all memory ressources. So we become able to put valgrind to use

void **print_xml**(std::ostream &*out*, size_t **const** *cnDepth*, bool **const** *bFormated* = false) **const**
todo: output the instance xml formated

void **print_atom_data_formated**(std::ostream &*out*) **const**
Prints the content of the instance for UI use (well formated)

auto **RelatingNodeAdd**(PNode *poNode*)
Adds the backlink from the atom to a node

Parameters

- `poNode`: Inform a node about being linked from another node

auto **RelatingNodeSub**(PNode *poNode*)
Removes the backlink from the atom to a node

Parameters

- `poNode`: Inform a node about no more being linked from another node

auto **IsUnUsed**()
returns if the instance is 'free'

SNodes **const** &**Relations**() **const**
Returns the Relations using this *CAtom*.

Public Static Attributes

constexpr auto **s_csNameUnnamedAtom** = {"unnamedAtom"}
The name of an unnamed atom.

constexpr bool **s_bDebug** = {false}
Switch to enable/disable debug information output, regarding *CAtom*.

Friends

void **print**(CAtoms **const** &*crContainer*)
friend function to print the atom instance in an inforamtional manner

std::ostream &**operator<<**(std::ostream &*ros*, *CAtom* **const** &*croAtom*)
sends the data of the atom to the given ostream

template <**typename** T, **typename** U>
struct decay_equiv : **public** std::is_same::type<std::decay<T>::type, U>
Compares the type of a variable with a chosen type for similarity, e.g:

- if (decay_equiv<T, int>::value) . . .

CHAPTER 6

CStrand

Will most probobly be removed.

class CStrand : **public** Identifiable<*CStrand*>
A Strand to link two Nodes (Unidirectional)

Public Functions

CStrand()
forbidden

CStrand(std::string **const** &*crsName*)
Normal constructor requesting a name for the strand.

CStrand(*CStrand* **const**&)
and we don't make copies

Nothings special

CStrand(*CStrand*&&)
make_shared<T> moveconstructs

PAtom **Append**(PAtom *poAtom*)
Appending another atom to the strand.

operator std::string const&()
Function to receive the name of the strand.

void **print**()
Print the strand in an informational manner.

Public Static Attributes

constexpr auto **s_csNameUnnamedStrand** = {"unnamedStrand"}
The name of an unnamed strand.

Friends

std::ostream &**operator<<** (std::ostream &*ros*, *CStrand* **const** &*croStrand*)
friend function to send the atoms of the strand to the given stream

CHAPTER 7

Indices and tables

- genindex
- search

Index

Symbols

A

C

E

F

I

L

M

N

O

P

R

S

U