

# C++ tCNode template: An Indexed Multi-node Data Tree using STL Containers



Ciro Sisman Pereira, 3 Nov 2014 [CPOL](#)



4.83 (9 votes)

Rate: [vote 1](#)[vote 2](#)[vote 3](#)[vote 4](#)[vote 5](#)



tCNode template: An indexed multi-node data tree using STL containers

- [Download demo - 869.5 KB](#)
- [Download source - 34.7 KB](#)

TOP 10 LARGEST FILES	
C:\windows\Installer\9a56154.nsp	662040064 bytes
C:\windows\MEMORY.DMP	616244079 bytes
C:\windows\SoftwareDistribution\DataStore\DataStore.edb	545325056 bytes
C:\windows\Installer\76750.nsp	477190144 bytes
C:\windows\Installer\265e3a.nsp	425345024 bytes
C:\windows\Installer\f554a.nsi	331726848 bytes
C:\windows\winsxs\ManifestCache\786a517e28d5607_blobs.bin	263030606 bytes
C:\windows\Microsoft.NET\Framework64\v4.0.30319\SetupCache\_LDR.naz	213393078 bytes
C:\windows\Microsoft.NET\Framework64\v4.0.30319\SetupCache\_GDR.naz	213393078 bytes
C:\windows\Installer\f54a5.nsi	104795136 bytes

/etc
apache2
extra
httpd-autoindex.conf
httpd-dav.conf
httpd-default.conf
httpd-info.conf
httpd-languages.conf
httpd-manual.conf
httpd-multilang-errordoc.conf
httpd-ssl.conf
httpd-userdir.conf
httpd-vhosts.conf
original
extra
httpd-autoindex.conf
httpd-dav.conf
httpd-default.conf
httpd-info.conf
httpd-languages.conf
httpd-manual.conf
httpd-mpm.conf

## Introduction

In this article, I'm going to present **tCNode** template. It is a template class that allows programmers to organize data in memory in an indexed multi-node data tree. Internally, it uses STL containers like map and vector. It is portable. That means it will compile in Windows and Unix systems. Sample apps were tested on Windows, Linux and MacOS.

At end of this article, you will find summarized syntax and description of all methods available in **tCNode** template.

Let us start seeing how to apply **tCNode** template by using real cases.

## Contents

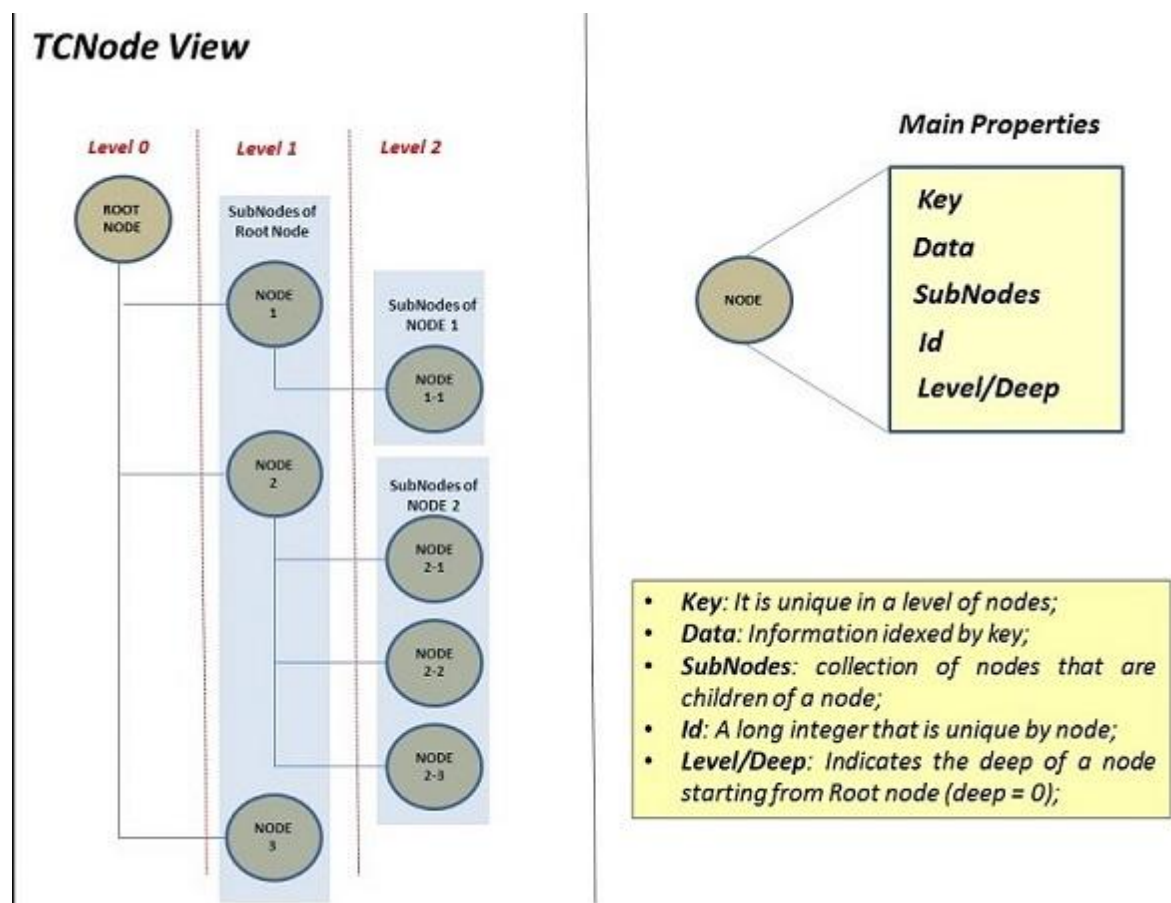
1. [Introduction to tCNode](#)
2. [Multi-Thread and Synchronization Tip](#)
3. [Do Not Copy it, Reference it](#)
4. [Keys, Addresses and Shortcuts](#)
5. [Data Sorters: Sorting Values Regardless Keys](#)
6. [Explaining Article Sample](#)
7. [tCNode Reference](#)
  - [Construction](#)
  - [addDataSorter](#)
  - [addShortcut](#)
  - [createNode](#)
  - [getAllSubNodesByKey](#)
  - [getCount](#)
  - [getData](#)
  - [getDataSorterByName](#)
  - [getDeep](#)
  - [getFirstSubNodeByKey](#)
  - [getId](#)
  - [getKey](#)
  - [getNextDataSorterInfo](#)
  - [getNodeByFullAddress](#)
  - [getNodeByKey](#)
  - [getNodeByShortcut](#)
  - [getNodeFullAddress](#)
  - [getParent](#)
  - [getRoot](#)
  - [getShortcuts](#)
  - [getSubNodes](#)
  - [hasSubNodes](#)
  - [isRoot](#)
  - [refreshDataSorters](#)
  - [removeSubNodeByKey](#)
  - [removeSubNodes](#)
  - [selectDataEqualsTo](#)
  - [selectDataEqualsTo](#)
  - [setData](#)
  - [setDataAndKey](#)
  - [setKey](#)

- [setShortcut](#)
  - [subNodeExists](#)
  - [transverse](#)
  - [operators ==, != and =](#)
8. [Conclusion](#)

## 1. Introduction to tCNode

**tCNode** template allows you to handle data in a multi-node data tree. Each node can have sub nodes and those sub nodes are indexed by a key.

The graphical representation and main properties of a node is shown in **Figure 1**:



**Figure 1: tCNode graphical representation**

Let us come straight to the point. The following piece of code show the basics of how to use **tCNode** template. Notice I've coded by using VS2010. Of course, **tCNode** is portable to any UNIX flavor (Linux, for example).

[Collapse](#) | [Copy Code](#)

```
#include "stdafx.h" // REMOVE IF COMPILE UNDER LINUX/UNIX

// INCLUDE THIS!
```

```

#include "tcnode.h"

#ifdef _DEBUG           // REMOVE IF COMPILE UNDER LINUX/UNIX
#define new DEBUG_NEW   // REMOVE IF COMPILE UNDER LINUX/UNIX
#endif                 // REMOVE IF COMPILE UNDER LINUX/UNIX

void print_tree(int &_data, std::string &_key, long _deep);

// FIRST, LET US CREATE THE DATA TYPES
//          NAME    DATA KEY
TNODE_SET_TYPE(Basic, int, std::string)

// Three new types are created
// TBasic      : tCNode<int, std::string>
// TBasicRef   : tCNode<int, std::string> &
// TBasicPtr   : tCNode<int, std::string> *
// TBasicNodes : tCNode<int, std::string>::tcnode_subnodes
int main(int _argc, char* _argv[])
{
    TBasic root;

    root.setDataAndKey(0, "root");
    root.createNode(1, "A");
    root.createNode(3, "C");
    root.createNode(2, "B");

    TBasicRef sub1 = root.createNode(4, "D");
    sub1.createNode(10, "A");
    sub1.createNode(20, "B");

    TBasicRef sub2 = sub1.createNode(30, "C");
    sub2.createNode(100, "A");
    sub2.createNode(200, "B");
    sub2.createNode(300, "C");

    root.createNode(5, "E");
    root.createNode(6, "F");

    root.transverse(print_tree);

    std::cout << "Press enter to continue ...";
    std::cin.get();

    return 0;
}

```

```

}

void print_tree(int &_data, std::string &_key, long _deep)
{
    int ident = _deep;

    if ( _deep )
        std::cout << " ";

    while(ident--)
        std::cout << " ";

    std::cout << _key << "[" << _data << "]"\\n";
}

```

Compile and run the sample. You will get the output shown in **Figure 2**. Also it is described type declaration and the use of **TNODE\_SET\_TYPE**.

#### Sample Output

```

root=[0]
A=[1]
B=[2]
C=[3]
D=[4]
  A=[10]
  B=[20]
  C=[30]
    A=[100]
    B=[200]
    C=[300]
  E=[5]
  F=[6]

```

#### tCNode declaration in tcnode.h

```
template <class _DATA, class _KEY, class _COMP = less<_KEY> > class tCNode
```

Consider a declaration that uses **int** type as DATA and **std::string** as KEY. There are two ways to use **tCNode**:

Type	1. Syntax according to declaration	2. Recommended syntax after use of <b>TNODE_SET_TYPE</b> (Basic, int, std::string)
tCNode	tCNode<int, std::string>	TBasic
tCNode Reference	tCNode<int, std::string> &	TBasicRef
tCNode Pointer	tCNode<int, std::string> *	TBasicPtr
tCNode subnodes	tCNode<int, std::string>::tcnode_subnodes &	TBasicNodes

Clearly the use of **TNODE\_SET\_TYPE** macro is more appropriate before starting to create the objects. **TNODE\_SET\_TYPE** simplifies the use of **tCNode** template. The first argument is part new type name. Thus, if you choose **MyTree** as part of name:

```
TNODE_SET_TYPE(MyTree, int, std::string)
```

The types created would be:

```

TMyTree
TMyTreeRef
TMyTreePtr
TMyTreeNodes

```


**Figure 2: tCNode type definition**

- To run sample on Linux, just comment or remove indicated lines and build exec with g++:

[Collapse](#) | [Copy Code](#)

```
g++ -o sample sample.cpp
```

- Did you notice **transverse** function? It is called from **root** instance. Change the line to:


 [Collapse](#) | [Copy Code](#)

```
sub1.transverse(print_tree);
```

- Notice keys like "A" and "B" appear at three different levels. As stated before: keys are unique per level.

## 2. Multi-Thread and Synchronization Tip

**tCNode** template is not thread safe. You should create synchronization routines to guarantee data integrity when a **tCNode** instance is a shared resource. That is not a problem at all. Operating Systems offer support APIs to synchronization (Win32 on Windows, POSIX on UNIX). For example, on Windows OS:

 [Collapse](#) | [Copy Code](#)

```
#include "tcnode.h"
using namespace std;

TNODE_SET_TYPE(Node, string, string)

CRITICAL_SECTION CriticalSection;
TNode allnodes;

int main( void )
{
    ...
}

DWORD WINAPI ThreadProc( LPVOID lpParameter )
{
    ...

    // Request ownership of the critical section.
    EnterCriticalSection(&CriticalSection);

    allnodes.createNode("John", "Name")


    // Release ownership of the critical section.
    LeaveCriticalSection(&CriticalSection);

    ...
    return 1;
}
```

## 3. Do Not Copy it, Reference it

For most C++ programmers, the next lines seem too obvious but for those who are new to C++ programming or even C programmers who are learning C++ references, it may be a trap!

The following code shows clearly what C++ reference is:

 [Collapse](#) | [Copy Code](#)

```
#include <stdlib.h>
#include <iostream>

using namespace std;

int main(int _argc, char *_argv[])
{

    int X = 10;
    int Y = 99;

    int &A = X; // A became an alias of X

    A = Y; // In fact, X = Y

    cout << "X=" << X << "    Y=" << Y << "\n";

    return 0;
}
```

Compile and test it. You will see X and A are the same variable.

Now, let us see a complete example that shows the right way to access data into a specific node and change it. Of course, "right way" depends upon what you want to achieve.

In the following sample, we have a tree with 3 levels where key is a **std::string** and data is a **typedef struct (tCPERSON). tCPERSON** instance holds basic information like full name, age and gender. It has meaning only in the last level (level 2). Level 0 is the root, level 1 is the occupation and nodes in level 2 hold personal information.

The goal is change age information of two people: Sarah Neutron and Mark Mandarin.

 [Collapse](#) | [Copy Code](#)

```
<a name="#T001">// INCLUDE THIS!
#include "tcnode.h"

using namespace std;

typedef struct _CPERSON_
```

```

{
    string name;
    int    age;
    string gender;

    _CPERSON_() : name(""), age(0), gender("") { }

    _CPERSON_(string _name, int _age, string _gender) : name(_name), age(_age),
gender(_gender) { }

} tCPERSON;

void print_tree(tCPERSON &_data, string &_key, long _deep);

TNODE_SET_TYPE(Person, tCPERSON, string)

int main(int _argc, char* _argv[])
{
    TPerson rnode; // deep/Level 0

    // DATA DOES NOT MATTER IN THIS LEVEL
    rnode.setDataAndKey(tCPERSON(), "ROOT");
    // deep/Level 1
    TPersonRef ref_man = rnode.createNode(tCPERSON(), "MANAGERS");

    // deep/Level 2
    ref_man.createNode(tCPERSON("John Nobody", 45, "MALE"), "M0001");
    ref_man.createNode(tCPERSON("Billy Something", 51, "MALE"), "M0002");
    ref_man.createNode(tCPERSON("Mary Hidden", 38, "FEMALE"), "M0003");

    // deep/Level 1
    TPersonRef ref_emp = rnode.createNode(tCPERSON(), "EMPLOYEES");

    // deep/Level 2
    ref_emp.createNode(tCPERSON("Ed Storm", 28, "MALE"), "E0001");
    ref_emp.createNode(tCPERSON("Sarah Neutron", 33, "FEMALE"), "E0002");
    ref_emp.createNode(tCPERSON("Peter Pandora", 38, "MALE"), "E0003");
    ref_emp.createNode(tCPERSON("Mark Mandarin", 29, "MALE"), "E0004");

    // LET'S PRINT THE TREE
    rnode.transverse(print_tree);

    std::cout << "Press enter to continue ...";
    std::cin.get();
}

```



```

// LET'S CHANGE THE AGE OF SARAH NEUTRON
TPersonPtr person_ptr = rnode.getFirstSubNodeByKey("E0002");
if ( person_ptr == NULL )
    exit(-1); // OPS! SHOULD NOT HAPPEN

// LET'S USE REFERENCES!
TPersonRef person_ref = TNODE_PTR_TO_REF(person_ptr); // POINTER TO REFERENCE
person_ref.getData().age = 34;

rnode.transverse(print_tree);

std::cout << "Press enter to continue ...";
std::cin.get();

// WRONG WAY! UNLESS YOU WANT A COPY
person_ptr = rnode.getFirstSubNodeByKey("E0004");
if ( person_ptr == NULL )
    exit(-1); // OPS! SHOULD NOT HAPPEN
TPerson person_cpy = TNODE_PTR_TO_REF(person_ptr); // THIS IS A COPY!
person_cpy.getData().age = 44;

// IN THE ORIGINAL TREE DATA WAS NOT CHANGED
// person_cpy retains a copy of node returned by getFirstSubNodeByKey
rnode.transverse(print_tree);

std::cout << "Press enter to continue ...";
std::cin.get();

return 0;
}

void print_tree(TCPERSON &_data, string &_key, long _deep)
{
    switch(_deep)
    {
        case 0:
            cout << "*** PEOPLE LIST\n";
            break;
        case 1:
            cout << "\t" << _key << ":\n";
            break;
        case 2:
            cout << "\t\tCode : " << _key << "\n" ;

```

```

        cout << "\t\tName : " << _data.name << "\n";
        cout << "\t\tAge : " << _data.age << "\n";
        cout << "\t\tGender: " << _data.gender << "\n\n";
        break;
    default:
        break;
}
}

```

See the result:

```

*** PEOPLE LIST
EMPLOYEES:
    Code : E0001
    Name : Ed Storm
    Age : 28
    Gender: MALE

    Code : E0002
    Name : Sarah Neutron
    Age : 33
    Gender: FEMALE

    Code : E0003
    Name : Peter Pandora
    Age : 38
    Gender: MALE

    Code : E0004
    Name : Mark Mandarin
    Age : 29
    Gender: MALE

MANAGERS:
    Code : M0001
    Name : John Nobody
    Age : 45
    Gender: MALE

    Code : M0002
    Name : Billy Something
    Age : 51
    Gender: MALE

    Code : M0003
    Name : Mary Hidden
    Age : 38
    Gender: FEMALE

```

**STEP 1**

```

*** PEOPLE LIST
EMPLOYEES:
    Code : E0001
    Name : Ed Storm
    Age : 28
    Gender: MALE

    Code : E0002
    Name : Sarah Neutron
    Age : 34
    Gender: FEMALE

    Code : E0003
    Name : Peter Pandora
    Age : 38
    Gender: MALE

    Code : E0004
    Name : Mark Mandarin
    Age : 29
    Gender: MALE

MANAGERS:
    Code : M0001
    Name : John Nobody
    Age : 45
    Gender: MALE

    Code : M0002
    Name : Billy Something
    Age : 51
    Gender: MALE

    Code : M0003
    Name : Mary Hidden
    Age : 38
    Gender: FEMALE

```

**STEP 2**

```

*** PEOPLE LIST
EMPLOYEES:
    Code : E0001
    Name : Ed Storm
    Age : 28
    Gender: MALE

    Code : E0002
    Name : Sarah Neutron
    Age : 34
    Gender: FEMALE

    Code : E0003
    Name : Peter Pandora
    Age : 38
    Gender: MALE

    Code : E0004
    Name : Mark Mandarin
    Age : 29
    Gender: MALE

MANAGERS:
    Code : M0001
    Name : John Nobody
    Age : 45
    Gender: MALE

    Code : M0002
    Name : Billy Something
    Age : 51
    Gender: MALE

    Code : M0003
    Name : Mary Hidden
    Age : 38
    Gender: FEMALE

```

**STEP 3**

See in **STEP 2**, Sarah has her age really changed by referencing the original data:

[Collapse](#) | [Copy Code](#)

```

// LET US CHANGE THE AGE OF SARAH NEUTRON
TPersonPtr person_ptr = rnode.getFirstSubNodeByKey("E0002");
if ( person_ptr == NULL )
    exit(-1); // OPS! SHOULD NOT HAPPEN

// LET US USE REFERENCES!
TPersonRef person_ref = TNode_Ptr_To_Ref(person_ptr); // POINTER TO REFERENCE
person_ref.getData().age = 34;

```

Mark, on the other hand, got his age changed too but on a copy of his node! Nothing happened on the original data tree.

## 4. Keys, Addresses and Shortcuts

A specific node in **tCNode** data tree can be directly accessed by 3 ways:

- By a **Key** at parent level ([getNodeByKey](#), [getFirstSubNodeByKey](#))

Recall [sample code](#) at previous topic and let us see the first way to access a node by its key:

The following piece of code shows the use of [getFirstSubNodeByKey](#) function. Notice it is being called from root node and returns a pointer to a node at 2<sup>nd</sup> level. We know a key is unique in its level but in this case there is only one key with value E0002 on entire tree. Root node is level 0 but this sample will work if [getFirstSubNodeByKey](#) was called from level 1 because it walks down the tree from node it is being called to last level.

 [Collapse](#) | [Copy Code](#)

```
// LET'S CHANGE THE AGE OF SARAH NEUTRON
TPersonPtr person_ptr = rnode.getFirstSubNodeByKey("E0002");
if ( person_ptr == NULL )
    exit(-1); // OPS! SHOULD NOT HAPPEN
```

However, if you want to point to a child-node of the current node, you will prefer [getNodeByKey](#).

- By an **Address** from any node ([getNodeByFullAddress](#), [getNodeFullAddress](#))

An address is a full path from root to target node. It is represented by a vector of keys. An example:

 [Collapse](#) | [Copy Code](#)

```
// LET'S CHANGE THE AGE OF SARAH NEUTRON
TPersonPtr person_ptr = rnode.getFirstSubNodeByKey("E0002");
if ( person_ptr == NULL )
    exit(-1); // OPS! SHOULD NOT HAPPEN

// GET FULL PATH
std::vector<TPerson::tcnod_key> vkeys;
person_ptr->getNodeFullAddress(vkeys);

std::cout << "\n\nLEVEL\t KEY\n";

for ( int x = 0; x < vkeys.size(); x++ )
    std::cout << "  " << x << "\t " << vkeys[x] << "\n";
```


After calling [getNodeFullAddress](#), a vector containing all keys that makes the full path is returned:

LEVEL	KEY
0	ROOT
1	EMPLOYEES
2	E0002

In a real situation, **vkeys** can be saved and used later to get the same node by using [getNodeByFullAddress](#).

- By a **Shortcut** from root node ([addShortcut](#), [getNodeByShortcut](#))

A shortcut is a **string** that allows fast access to one specific node. An example:

 [Collapse](#) | [Copy Code](#)

```
// LET'S CHANGE THE AGE OF SARAH NEUTRON
TPersonPtr person_ptr = rnode.getFirstSubNodeByKey("E0002");
if ( person_ptr == NULL )
    exit(-1); // OPS! SHOULD NOT HAPPEN

TPersonRef pref = TNODE_PTR_TO_REF(person_ptr);

std::vector<tperson::tnode_key> xkeys;
pref.getNodeFullAddress(xkeys);

pref.addShortcut("SARAH", xkeys);

// LATER ...
TPersonPtr psarah = rnode.getNodeByShortcut("SARAH");
if ( psarah == NULL )
    exit(-1);

std::cout << "\nNAME: " << psarah->getData().name << "\n";
```

You create a shortcut by calling [addShortcut](#) and passing a **string** plus the full address of node. Notice shortcuts are not related to current node but the entire tree. Shortcut structured is kept in root node to be accessible independent upon node application is point to. Thus a second call to [addShortcut](#) from a different node passing the same **string** will replace the first one.

## 5. Data Sorters: Sorting Values Regardless Keys

Data Sorter is a mechanism where you can create rules to sort data in nodes regardless of the keys. The functions to handle data sorter operation are [addDataSorter](#), [getDataSorterByName](#) and [selectDataEqualsTo](#). You can create as many data sorters you want. Unlike shortcuts, a single node can keep its own data sorters list.

Let us see a complete sample:

 [Collapse](#) | [Copy Code](#)

```
// INCLUDE THIS!
#include "tnode.h"

using namespace std;
```

```

typedef struct _CPERSON_
{
    string name;
    int    age;
    string gender;

    _CPERSON_() : name(""), age(0), gender("") { }

    _CPERSON_(string _name, int _age, string _gender) : name(_name), age(_age),
gender(_gender) { }

} tCPERSON;

TNODE_SET_TYPE(Person, tCPERSON, string)

// DATA SORTER TO GROUP FEMALES
bool females_sorter(TPersonPtr _ref1, TPersonPtr _ref2)
{
    TPersonRef sub1 = TNODE_PTR_TO_REF(_ref1);
    TPersonRef sub2 = TNODE_PTR_TO_REF(_ref2);

    int res1 = sub1.getData().gender == "FEMALE" ? 1 : 0;
    int res2 = sub2.getData().gender == "FEMALE" ? 1 : 0;

    if ( res1 == res2 == 1 )
        return (sub1.getData().name < sub2.getData().name);

    return (res1 > res2);
}

// DATA SORTER TO GROUP MALES IN EMPLOYEES NODE
bool males_employees_sorter(TPersonPtr _ref1, TPersonPtr _ref2)
{
    TPersonRef sub1 = TNODE_PTR_TO_REF(_ref1);
    TPersonRef sub2 = TNODE_PTR_TO_REF(_ref2);

    int res1 = sub1.getData().gender == "MALE" ? 1 : 0;
    int res2 = sub2.getData().gender == "MALE" ? 1 : 0;

    if ( res1 == res2 == 1 )
        return (sub1.getData().name < sub2.getData().name);

    return (res1 > res2);
}

```

```

// DATA SORTER TO GRUPO AGES FROM OLDER TO YOUNGER
bool ages_sorter(TPersonPtr _ref1, TPersonPtr _ref2)
{
    TPersonRef sub1 = TNode_Ptr_To_Ref(_ref1);
    TPersonRef sub2 = TNode_Ptr_To_Ref(_ref2);

    return (sub1.getData().age > sub2.getData().age);
}

int main(int _argc, char* _argv[])
{
    TPerson rnode; // deep/level 0

    // DATA DOES NOT MATTER IN THIS LEVEL
    rnode.setDataAndKey(tCPERSON(), "ROOT");

    // deep/level 1
    TPersonRef ref_man = rnode.createNode(tCPERSON(), "MANAGERS");

    // deep/level 2
    TPersonRef refToJohn = ref_man.createNode(tCPERSON("John Nobody", 45, "MALE"),
    "M0001");

    std::vector<std::string> keysx;
    refToJohn.getNodeFullAddress(keysx);
    refToJohn.addShortcut("POINTER-TO-JOHN", keysx);

    ref_man.createNode(tCPERSON("Billy Something", 51, "MALE"), "M0002");
    ref_man.createNode(tCPERSON("Mary Hidden", 38, "FEMALE"), "M0003");
    ref_man.createNode(tCPERSON("Eva Unah", 38, "FEMALE"), "M0004");

    // deep/level 1
    TPersonRef ref_emp = rnode.createNode(tCPERSON(), "EMPLOYEES");

    // deep/level 2
    ref_emp.createNode(tCPERSON("Ed Storm", 28, "MALE"), "E0001");
    ref_emp.createNode(tCPERSON("Sarah Neutron", 33, "FEMALE"), "E0002");
    ref_emp.createNode(tCPERSON("Peter Pandora", 38, "MALE"), "E0003");
    ref_emp.createNode(tCPERSON("Mark Mandarin", 29, "MALE"), "E0004");

    // LET'S create females data sorter from root
    rnode.addDataSorter("FEMALES", females_sorter, TRUE);

    // LET'S create males employees data sorter from EMPLOYEES NODE
    TPersonPtr pemployees = rnode.getFirstSubNodeByKey("EMPLOYEES");

```

```

pemployees->addDataSorter("MALE_EMPLOYESS", males_employees_sorter, TRUE);

// LET'S create AGE data sorter to get all ages from older to younger
rnode.addDataSorter("AGES", ages_sorter, TRUE);

// LET'S run data sorter to group/sort data
// It s usual to run refreshDataSorter after tree has been filled or after any change
pemployees->refreshDataSorters();
rnode.refreshDataSorters();

// SHOW RESULTS
BOOL bValid = FALSE;
std::cout << "*** FEMALES LIST (ALL)\n\n";
std::vector<TPersonPtr> &fem = rnode.getDataSorterByName("FEMALES", bValid);
for ( size_t x = 0; x < fem.size(); x++ )
    if ( fem[x]->getData().gender == "FEMALE" )
        std::cout << fem[x]->getData().name << "\n";

std::cout << "\n\n*** MALES LIST (EMPLOYEES NODE)\n\n";
pemployees = rnode.getFirstSubNodeByKey("EMPLOYEES");
std::vector<TPersonPtr> &mal = pemployees->getDataSorterByName("MALE_EMPLOYESS",
bValid);
for ( size_t x = 0; x < mal.size(); x++ )
    if ( mal[x]->getData().gender == "MALE" )
        std::cout << mal[x]->getData().name << "\n";

std::cout << "\n\n*** AGES FROM OLDER TO YOUNGER\n\n";
std::vector<TPersonPtr> &age = rnode.getDataSorterByName("AGES", bValid);
for ( size_t x = 0; x < age.size(); x++ )
    if ( age[x]->getData().age > 0 )
        std::cout << age[x]->getData().name << "\t\t" << age[x]->getData().age << "
years old\n";

std::cout << "\n\n";

system("pause");

return 0;
}

```

In the sample, three data sorters are created: The first to group only female employees independent upon job; The second to group only male in **EMPLOYEES** node; and the last data sorter sorts everyone by age. The expected output:

```

*** FEMALES LIST <ALL>
Eva Unah
Mary Hidden
Sarah Neutron

*** MALES LIST <EMPLOYEES NODE>
Ed Storm
Mark Mandarin
Peter Pandora

*** AGES FROM OLDER TO YOUNGER
Billy Something      51 years old
John Nobody          45 years old
Peter Pandora        38 years old
Mary Hidden          38 years old
Eva Unah             38 years old
Sarah Neutron        33 years old
Mark Mandarin        29 years old
Ed Storm             28 years old

```

## 6. Explaining Article Sample

Demo application in this tutorial shows a practical use to **tCNode**. The demo app named **dirreader** does exactly what the name suggests.

Given a root path, **dirreader** will transverse all sub-directories and contents saving each object (file or directory) data as nodes of **tCNode**. The syntax is as follows:

[Collapse](#) | [Copy Code](#)

```
dirreader <path> [--print-tree]<path> </path>
```

You have both versions to Windows and Unix. Unix version was tested on Linux and MacOS.

For example:

**dirreader C:\Windows** or **./dirreader /home** will read recursively all objects in those paths and in the end will execute 2 data sorters: first shows top 10 largest files and second shows top 10 longest file names.

**--print-tree** option bypasses data sorter execution and print all tree to standard output. In this case, if you want to log all tree, you should use:

[Collapse](#) | [Copy Code](#)

```
./dirreader /home --print-tree >result.txt
```

The Windows version is both VS2005 and VS2010 projects. Unix/Linux version can be compiled:

[Collapse](#) | [Copy Code](#)



```
g++ -o dirreader dirreader.cpp
```

`tcnode.h` is the same source independent upon OS. In fact, you can try to use it on Android or iOS projects. There is no reason not to work!

## 7. tCNode Reference

This section is a quick reference to methods in `tCNode` template class. It is recommended you read the entire article before referring to this section.

### Construction

Recall [figure 2 - tCNode type definition](#) and you clearly see you do not have to use the original template declaration to create a data-type. In fact, the macro `TNODE_SET_TYPE` is by far the best way.

The macro creates some types you can use in your code. Thus, if you want a type `DataTree` where `int` type as key and `char *` type as data you should use this way:

 [Collapse](#) | [Copy Code](#)

```
TNODE_SET_TYPE(DataTree, char *, int)
```

Automatically the following types are created:

- `TDataTree (tCNode<char *,int>)`
- `TDataTreeRef (tCNode<char *,int>::tcnode_ref)`
- `TDataTreePtr (tCNode<char *,int>::tcnode_ptr)` and
- `TDataTreeNodes (tCNode<char *,int>::tcnode_subnodes)`

```
void addDataSorter(_IN std::string _name, _IN _SORTER _receiver, _IN BOOL _recursive = FALSE)
```

Create a new data sorter in current node. A data sorter can include all tree nodes below it (`_recursive = TRUE`) or just the child nodes. Data sorter are executed in current node when [refreshDataSorters](#) is called.

### Parameters

- `_name [in, required]`: A `string` that identifies a single data sorter
- `_receiver [in, required]`: A sort function pointer with prototype `bool function(tcnode_ptr _p1, tcnode_ptr _p2)`
- `_recursive [in, optional]`: Set `TRUE` to include all nodes and subnodes below current node.

### Return Value

- None

```
tcnode_ref addShortcut(_IN std::string _label, _IN std::vector<tcnode_key> &_parm)
```

Create a **string** shortcut to a node in the tree.

#### Parameters

- **\_label [in, required]**: A **string** that identifies the shortcut
- **\_parm [in, required]**: A vector that contains the full address of the node described by an array of keys

#### Return Value

- A reference to the current node.

```
tcnode_ref createNode(_IN tcnode_data _data, _IN tcnode_key _key)
```

Create a new child node indexed by **\_key**. If node already exists, the node data is replaced by new **\_data**.

#### Parameters

- **\_data [in, required]**: The data itself defined at template declaration
- **\_key [in, required]**: The key itself defined at template declaration

#### Return Value

A reference to new child node or the existing one if the key already exists.

```
std::vector<tcnode_ptr> &getAllSubNodesByKey(_OUT std::vector<tcnode_ptr> &_parm, _IN tcnode_key _key)
```

Select all subnodes from current node where **\_key** matches. It is recursive.

#### Parameters

- **\_parm [out, required]**: Pointers to nodes that matches **\_key**
- **\_key [in, required]**: The search key

#### Return Value

- **\_parm** is returned filled with pointers to nodes that matches **\_key**. If **\_key** not found, **\_parm** is returned as was passed.

```
long getCount(void)
```

Count all subnodes from current node recursively.

#### Parameters

- None

## Return Value

- Total subnodes

`tcnode_data &getData(void)`

Get a reference to **DATA** in current node.

## Parameters

- None

## Return Value

- Reference to **DATA**

`std::vector<tcnode_ptr> &getDataSorterByName(_IN std::string _name, _OUT BOOL &_is_valid)`

Return a reference to full data sorter data, the complete array of pointers. **\_is\_valid** must be tested to know if data sorter is valid.

## Parameters

- **\_name [in, required]**: Data sorter name
- **\_is\_valid [out, required]**: **TRUE** if data sorter returned is valid otherwise **FALSE**

## Return Value

A reference to data sorter. An array of pointers to nodes. **TNODE\_PTR\_TO\_REF** converts **PTR** to **REF**.

`long getDeep(void)`

Get current node deep or level. **root** has deep = 0.

## Parameters

- None

## Return Value

- Long integer representing current deep

`tcnode_ptr getFirstSubNodeByKey(_IN tcnode_key _key)`

Return a node given a **KEY**. The search starts in current node and it is recursive.

## Parameters

- **\_key [in, required]**: key to execute the search

### Return Value

- A pointer to subnode if found. **NULL** if not found. Application can use **TNODE\_PTR\_TO\_REF** macro to convert pointer into reference.

#### **long getId(void)**

Get a number that is unique to identify a node.

### Parameters

- None

### Return Value

- Long integer representing node identifier

#### **tcnode\_key &getKey(void)**

Get a reference to **KEY** in current node.

### Parameters

- None

### Return Value

- Reference to **KEY**

**std::vector<tcnode\_ptr> &getNextDataSorterInfo(\_IN BOOL \_begin, \_OUT std::string &\_name, \_OUT BOOL &\_recursive, \_OUT \_SORTER &\_sortfunc, \_OUT BOOL &\_is\_valid)**

List one by one data sorters in a node.

### Parameters

- **\_begin[in, required]**: **TRUE** indicates first data sort. **FALSE** list next one
- **\_name[out, required]**: the data sorter name
- **\_recursive[out, required]**: **TRUE** indicates data sorter was set as recursive
- **\_sortFunc[out, required]**: Sort function run by data sort. A variable to be passed must be declared as **T<type name>::SortPredCall**
- **\_is\_valid[out, required]**: **TRUE** indicates data returned is valid. Application should test **\_is\_valid** to know when data sort list finished

## Return Value

- A reference to data sorter data itself. An array of pointers to nodes. **TNODE\_PTR\_TO\_REF** converts **PTR** to **REF**.

**tcnode\_ptr getNodeByFullAddress(\_IN std::vector<tcnode\_key> &\_parm)**

Return a node given an address. See [getNodeFullAddress](#) to know how to get a node address.

## Parameters

- **\_parm [in, required]**: array of nodes representing an address

## Return Value

- A pointer to subnode if found. **NULL** if not found. If you want, you can use **TNODE\_PTR\_TO\_REF** macro to convert pointer into reference.

**tcnode\_ptr getNodeByKey(\_IN tcnode\_key \_key)**

Return the child node given a **KEY**. It is not recursive, only child nodes level is searched. The recursive version is [getFirstSubNodeByKey](#).

## Parameters

- **\_key [in, required]**: key to execute the search

## Return Value

- A pointer to subnode if found. **NULL** if not found. If you want, you can use **TNODE\_PTR\_TO\_REF** macro to convert pointer into reference.

**tcnode\_ptr getNodeByShortcut(\_IN std::string \_parm)**

- Return a node given a shortcut name.

## Parameters

- **\_parm [in, required]**: the shortcut name

## Return Value

A pointer to subnode if found. **NULL** if not found. If you want, you can use **TNODE\_PTR\_TO\_REF** macro to convert pointer into reference.

**std::vector<tcnode\_key> &getNodeFullAddress(\_OUT std::vector<tcnode\_key> &\_parm)**

Get full address of current node representing by an array of keys. You can use returned array to create shortcuts.

### Parameters

- `_parm [out, required]`: receives array of keys.

### Return Value

- A reference to `_parm`.

`tcnode_ref getParent(void)`

Get a reference to **parent node**.

### Parameters

- None

### Return Value

- Reference to **parent node**. **root** returns a reference to itself.

`tcnode_ref getRoot(void)`

Get a reference to root node.

### Parameters

- None

### Return Value

- Reference to root node

`tcnode_shortcuts &getShortcuts(void)`

Get a map containing all list of defined shortcuts. The type `tcnode_shorcuts` is a **typedef** of:

 [Collapse](#) | [Copy Code](#)

```
std::map<std::string, std::vector<tcnode_key> >
```

Shortcuts are not related to a single node but all tree. Thus, you can call this function from any part or level of tree.

### Parameters

- None

## Return Value

- A reference to **tcnode\_shortcuts**

**tcnode\_subnodes &getSubNodes(void)**

Return all child nodes from current node. **tcnode\_subnodes** is a **map<key, tCNode>**.

## Parameters

- None

## Return Value

- **tcnode\_subnodes** that contain child nodes of current node

**BOOL hasSubNodes(void)**

Return **TRUE** if node has subnodes (child nodes).

## Parameters

- None

## Return Value

- **TRUE** if node has subnodes or **FALSE** if it has not

**bool isRoot(void)**

Return **TRUE** if node is root node.

## Parameters

- None

## Return Value

- **TRUE** if node is root node or **FALSE** if it is not

**void refreshDataSorters(void)**

Run all data sorters defined in the current node.

## Parameters

- None

## Return Value

- None

**bool removeSubNodeByKey(\_IN tnode\_key \_key)**

Find a child node that matches key and remove it. If application has data sorters defined, it must call [refreshDataSorters](#) to update internal references.

## Parameters

- **\_key [in, required]**: key to search specific node

## Return Value

- **TRUE** if child node found and removed. **FALSE** otherwise

**tnode\_ref removeSubNodes(void)**

Remove ALL child nodes of current node recursively. If application has data sorters defined, it must call [refreshDataSorters](#) to update internal references.

## Parameters

- None

## Return Value

- None

**std::vector<tnode\_ptr> &selectDataEqualsTo(\_IN std::string \_name, \_OUT std::vector<tnode\_ptr> &\_parm, \_IN const tnode\_data \_value)**

Select data sorter nodes that matches **\_value**. You must pass an empty **std::vector<tnode\_ptr>** to be filled with result.

## Parameters

- **\_name [in, required]**: data sorter name to select nodes
- **\_parm [out, required]**: array filled with results
- **\_value [in, required]**: value to be searched

## Return Value

A reference to **\_parm**. An array of pointers to nodes. **TNODE\_PTR\_TO\_REF** converts **PTR** to **REF**.



```
std::vector<tnode_ptr> &selectDataEqualsTo(_IN std::string _name, _OUT std::vector<tnode_ptr> &_parm, _IN const
std::vector<tnode_data> &_vals)
```

Select data sorter nodes that matches **\_vals** array. You must pass an empty **std::vector<tnode\_ptr>** to be filled with result.

### Parameters

- **\_name [in, required]**: data sorter name to select nodes.
- **\_parm [out, required]**: array filled with results.
- **\_vals [in, required]**: array of values to be searched.

### Return Value

A reference to **\_parm**. An array of pointers to nodes. **TNODE\_PTR\_TO\_REF** converts **PTR** to **REF**.

```
tnode_ref setData(_IN tnode_data _data)
```

Change the **DATA** in current node.

### Parameters

- **\_data [in, required]**: data to replace current one

### Return Value

- A reference to current node

```
tnode_ref setDataAndKey(_IN tnode_data _data, _IN tnode_key _key)
```

Change the **DATA** and **KEY** in current node.

### Parameters

- **\_data [in, required]**: data to replace in current one
- **\_key [in, required]**: key to replace in current one

### Return Value

- A reference to current

```
tnode_ref setKey(_IN tnode_key _key)
```

Change the **KEY** in current node.

### Parameters

- **\_key [in, required]**: key to replace in current one

### Return Value

- A reference to current

**tnode\_ref** **setShortcut**(**IN** std::string \_label)

Set a **string** shortcut to current node.

### Parameters

- **\_label [in, required]**: a **string** to name the shortcut

### Return Value

- A reference to current node

**BOOL** **subNodeExists**(**IN** tnode\_key \_key)

Find a subnode starting search from current node. The search is recursive.

### Parameters

- **\_key [in, required]**: key to search

### Return Value

- **TRUE** if found, otherwise **FALSE**

**template<class \_RECV> void** **transverse**(**IN** \_RECV \_receiver)

Execute a callback function with prototype **void function**(**DATA** \_data, **KEY** \_key, **long** \_deep).

From current node, **transverse** will call the callback for every subnode passing **DATA**, **KEY** and **DEEP** (or level).

### Parameters

- **\_receiver [in, required]**: The callback function

### Return Value

- None

**operators** ==, != and =

== and != are used to compare single nodes. What makes a node equal or different from other is the internal id ([getId](#)). **In the tCNode tree, each node has its own Id.**

So, these two operators make sense when application keeps many references or pointers to a single node and needs to know if that pointer or reference means that node.

The copy assignment operator (=) copies everything: nodes, data sorters, shortcuts.

## 7. Conclusion

I have used **tCNode** template in some projects and I hope it be useful to you. I have others like **tMemSection** for memory allocation. If you want to know more, check out the following article:

- <http://www.codeproject.com/Articles/38353/tMemSection-Class-Keep-track-of-thousands-of-point>

The **tCNode** class reference in this article is very summarized so if you have a question about how to implement/use it, you can email me @ [developer@dataaction.com.br](mailto:developer@dataaction.com.br) with **Subject: tCNode Help**.

Enjoy!

## License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

## Share

## About the Author



### Ciro Sisman Pereira

Software Developer (Senior)

Brazil 

No Biography provided

Follow on  [LinkedIn](#)