
Subject: NEW: Tree<T> container

Posted by [kohait00](#) on Fri, 30 Jul 2010 06:05:53 GMT

[View Forum Message](#) <> [Reply to Message](#)

hi folks, what about a Tree container? there is still none in U++, though there is a Link<T> with which one easily could implement binary trees (what still should be done anyway, they are quite usefull). But for trees with variable numbers of elements there is nothing. so here comes a first shot. what do you think of the following:

```
///  
  
template <class T>  
class Tree  
: protected Array<T>  
{  
protected:  
    typedef Array<T> B;  
    T * parent;  
  
public:  
    T *GetPtr() { return (T *) this; }  
    const T *GetPtr() const { return (const T *) this; }  
    T *GetParent() { return parent; }  
    const T *GetParent() const { return parent; }  
  
// Array interface  
  
    T& Add() { T & t = B::Add(); t.parent = (T *)this; return t; }  
    void Add(const T& x) { T & t = B::Add(DeepCopyNew(x)); t.parent = (T *)this; } // return t;  
}  
    void AddPick(pick_ T& x) { T & t = B::Add(new T(x)); t.parent = (T *)this; } // return t; }  
    T& Add(T *newt) { ASSERT(newt->parent == NULL); T & t = B::Add(newt); t.parent =  
(T *)this; return t; }  
  
    using B::operator[];  
    using B::GetCount;  
    using B::IsEmpty;  
  
    using B::Trim;  
    void SetCount(int n) { B::SetCount(n); for(int i = 0; i < B::GetCount(); i++)  
B::operator[](i).parent = (T *)this; }  
    void SetCountR(int n) { B::SetCountR(n); for(int i = 0; i < B::GetCount(); i++)  
B::operator[](i).parent = (T *)this; }  
    using B::Clear;  
  
    using B::Remove;
```

```

T&    Insert(int i)          { T & t = B::Insert(i); t.parent = (T *)this; return t; }
void  InsertPick(int i, pick_ T& x) { x.parent = (T *)this; B::InsertPick(i, x); }

using B::GetIndex;
using B::Swap;
using B::Move;

T      *Detach(int i)        { T *t = B::Detach(i); t->parent = NULL; return t; }
T&     Set(int i, T *newt)    { ASSERT(newt->parent == NULL); T & t = B::Set(i, newt); parent =
(T *)this; return t; }
void   Insert(int i, T *newt) { ASSERT(newt->parent == NULL); B::Insert(i, newt); newt->parent
= (T *)this; }

using B::Drop;
using B::Top;

T      *PopDetach()          { T * t = B::PopDetach(); t->parent = NULL; return t; }

void   Swap(Tree& b)          { B::Swap(b); for(int i = 0; i < b.GetCount(); i++) b[i].parent = (T
*)this; for(int i = 0; i < B::GetCount(); i++) B::operator[](i).parent = &b; }

Tree& operator<<(const T& x)    { Add(x); return *this; }
Tree& operator<<(T *newt)       { Add(newt); return *this; }
Tree& operator| (pick_ T& x)    { AddPick(x); return *this; }

// Array Interface end

Tree()
: parent(NULL)
{}

private:
Tree(const Tree&);
void operator=(const Tree&);

public:
#ifdef _DEBUG
void Dump() {
for(int i = 0; i < GetCount(); i++)
LOG((*this)[i]);
LOG("-----");
}
#endif
};

//

```

the Tree is actually a partially hidden Array of the same type elements, with a parent pointer. some methods from Array are free to access, some are overblended. the protected inheritance ensures that the overblended base methods are inaccessible. some methods are not critical though and can be exposed, in some, the parent ref is to be ensured. this thing can be thought in Vector and the Map flavours as well. in general i might need to a extended templated version where to specify which container to use. but this will grow in complicity at the beginning.

i'll try to provide a binary tree idea.

please post enhancements . i have not tested the whole thing very much though, first wanted to get sure that the model is right.

cheers
