Subject: Re: Ptr improve
Posted by tojocky on Fri, 20 May 2011 06:32:07 GMT
View Forum Message <> Reply to Message

kohait00 wrote on Thu, 19 May 2011 20:09here is a proposal..

```
//shared pointer
//idea borrowed from boost shared_ptr, an additional chunk of memory is managed
//which centrally holds the refcount of that object pointed to
//if Shared is created freshly, it AtomicInc's the ref count to 1;
//if a Shared is destroyed it AtomicDec's the refcount, and if its 0,
// it will delete both, the object and the refcount chunk
//if another instance is created as copy, the refcount is taken and incremented.
//if it is assigned, it decrements own existing counter, possibly releasing mem, and retains new
//pick semantic is not needed here anymore, it not even is possible
//since an 'operator=(const Shared<>&) is needed to aquire the source. pick is const in some
cases as well)
//thus Shared is only Moveable, without deepcopyoption, which in fact would speak agains the
idea of Shared anyway
//Attach / Detach remains

template <class T>
class Shared : Moveable< Shared<T> > {
 mutable T  *ptr;
 Atomic     *rfc;

 void Retain() const { ASSERT(rfc); AtomicInc(*rfc); }
 void Release()      { ASSERT(rfc); if(AtomicDec(*rfc) == 0) { Free(); delete rfc; rfc = NULL; } }

 void       Free()                 { if(ptr && ptr != (T*)1) delete ptr; }
 void       Chk() const              { ASSERT(ptr != (T*)1); }
 void       ChkP() const             { Chk(); ASSERT(ptr); }

public:
 void       Attach(T *data)          { Free(); ptr = data; }
 T         *Detach() pick_           { ChkP(); T *t = ptr; ptr = NULL; return t; }
 T         *operator-() pick_        { return Detach(); }
 void       Clear()                { Free(); ptr = NULL; }

 void       operator=(T *data)        { Attach(data); }
 void       operator=(const Shared<T>& d){ Release(); ptr = d.ptr; rfc = d.rfc; Retain(); }
 void       operator=(pick_ One<T>& d) { Attach(d.Detach()); }

 const T    *operator->() const        { ChkP(); return ptr; }
 T         *operator->()             { ChkP(); return ptr; }
 const T    *operator~() const         { Chk(); return ptr; }
 T         *operator~()              { Chk(); return ptr; }
```

```
 const T&    operator*() const          { ChkP(); return *ptr; }
 T&          operator*()                { ChkP(); return *ptr; }

 template <class TT>
 TT&         Create()                   { TT *q = new TT; Attach(q); return *q; }
 T&          Create()                   { T *q = new T; Attach(q); return *q; }

 bool        IsEmpty() const            { Chk(); return !ptr; }

 operator bool() const                  { return ptr; }

 Shared()                             { ptr = NULL; rfc = new Atomic(1); }
 Shared(T *newt)                        { ptr = newt; rfc = new Atomic(1); }
 Shared(const Shared<T>& p)             { ptr = p.ptr; rfc = p.rfc; Retain(); }
 ~Shared()                            { Release(); }

 Shared(pick_ One<T>& p)                { ptr = p.Detach(); rfc = new Atomic(1); }
 Shared(const One<T>& p, int)           { ptr = DeepCopyNew(*p); rfc = new Atomic(1); }
};
```

i first thought deriving from One<> but it will have problems with pick semantics
so i decided to stay with a clean separated version, but it's 80% One<> code
i added a convenience pick semantic for One<>

it's open for discussion..

```
Shared<Size> Test(Shared<Size> s)
{
 if(!s.IsEmpty())
  RLOG(*s);
 return s;
}

CONSOLE_APP_MAIN
{
 Shared<Size> p;
 {
 Shared<Size> s;

 s.Create();
 *s = Size(123,456);

 Shared<Size> q;
 q = Test(s);

 p = q;
```

```
 }
 if(!p.IsEmpty())
  RLOG(*p);
 One<Size> os;

 os.Create();
 *os = Size(1,2);
 p = os;
 RLOG(*p);

 os.Create();
 *os = Size(3,4);
 p = Shared<Size>(os);
 RLOG(*p);
}
```

Very nice,

I thought deriving from PtrBase, but it conflicts with Ptr<>.
Your proposal seems to be very clear.
