Subject: why no 'Ctrl* Ctrl::Clone() const = 0' (virtual constructor)
Posted by kohait00 on Tue, 31 Aug 2010 09:11:36 GMT
View Forum Message <> Reply to Message

hi guys

why does Ctrl not have Clone method? this would make possible to generic clone a bunch of controls (with some additioanl helpers) without knowing their type (i.e. control factory that is cloned)

```
Ctrl* Ctrl::Clone() const = 0;

//i.e. EditValue
Ctrl* EditValue::Clone() const
{
EditField * pc = new EditField();
pc->SetData(GetData());
pc->SetStyle(style);
pc->SetFont(font);
//etc those specific things
return pc;
}
```

this is a step towords a MVC like xml specifiable/parsable object inspector, which can be cloned itself..

Subject: Re: why no 'Ctrl* Ctrl::Clone() const = 0' (virtual constructor)
Posted by andrei_natanael on Tue, 31 Aug 2010 10:22:15 GMT
View Forum Message <> Reply to Message

Hi Konstantin,

Where will memory be freed? One will have to remember to free it and that's somehow against U++ rule: use pointers to point things not to manage heap.

Controls have specific methods and callbacks, so i don't see any reason to have a pointer to a Ctrl without knowing it's type (derived). It will be a "useless" Ctrl which accept sizing and positioning of it and some set/get data operations.

Depend on your usage it may be simple to create a generic Clone function not a method in Ctrl class. Ctrl is meant to be easily inherited without forcing one to overwrite(or define) unwanted methods. IMO, in case that'll be ever implemented it should be in Ctrl, with the possibility to be overwrite in derived class.

Andrei

Subject: Re: why no 'Ctrl* Ctrl::Clone() const = 0' (virtual constructor)
Posted by kohait00 on Tue, 31 Aug 2010 11:45:47 GMT
View Forum Message <> Reply to Message

the new Ctrl would go to an Array<Ctrl>::Add(Ctrl* newt);
so it wont disappear..

this clone feature is quite well known from C# and i learned to like it..it enables you to manage your object containers from 'bottom', kind of flexibility.

something like the following woule be a layer to use maybe, but includes changes as well

```
template<class T>
class Clonable
{
public:
 virtual T* Clone() const { return DeepCopyNew<T>(*(T*)this); }
 virtual T* PartialClone() const { return new T(); }
};

class ValueC
 : public Value, public Clonable<ValueC>
{
public:
 ValueC() {}
 ValueC(const Value& v) : Value(v) {}
};

template<class T>
T* Clone(const T& c) { return DeepCopyNew(c); }

template<class T>
T* PartialClone(const T&) { return new T(); }
```

Subject: Re: why no 'Ctrl* Ctrl::Clone() const = 0' (virtual constructor)
Posted by kohait00 on Thu, 09 Sep 2010 08:56:46 GMT
View Forum Message <> Reply to Message

i tried it with the following approach, it works in MSC9 but crashed in TDMGCC, any idea why? (maybe the brute cast is inapropriate..alignment of vtable wrong or sth..)

```
//copyable interface, implementing the Copy function, used by PolyDeepCopyNew
template<class T, class B = EmptyClass>
```

```
class Copyable : public B
{
public:
 virtual ~Copyable() {}
 virtual T* Copy() const { return new T(); }
};

//this is a nice helper, meant to be used like i.e. PolyCopying<EditCtrl>
//assigning Copy'ed instances to PolyCopying<Ctrl>* with a cast, as Ctrl is direct base class of EditCtrl
template<class T>
class PolyCopying : public Copyable< PolyCopying<T>, PolyDeepCopyNew<PolyCopying<T>, T> > {};

////

 PolyCopying<EditInt> abc;
 PolyCopying<Ctrl>* pabc = (PolyCopying<Ctrl>*)abc.Copy();
 PolyCopying<Ctrl>* pabc2 = pabc->Copy();

 Add(pabc2->HSizePos().TopPos(0,100)); //crashes here
 delete pabc2;
```

thats basicly why i need to have the Copy function in Ctrl..it makes factory cloning easy..

Subject: Re: why no 'Ctrl* Ctrl::Clone() const = 0' (virtual constructor)
Posted by kohait00 on Thu, 09 Sep 2010 09:54:08 GMT
View Forum Message <> Reply to Message

SOLUTION FOUND: by providing a copy interface with base class info

```
//this one works by providing additional information about the common base class
#if 1
//copyable interface defining the common base class C, without implementing Copy

template<class C>
class CopyableC
{
public:
 virtual ~CopyableC() {}
 virtual CopyableC* Copy() const = 0;
 virtual const C& GetC() const = 0;
 virtual C& GetC() = 0;

 operator const C&() const {return GetC(); }
```

```
 operator C&() {return GetC(); }
};

//provides the implementation of Copy, which is used by PolyDeepNew
//and implements the base class accessors.
//T is the derived type, i.e. EditInt, C is common base class, i.e. Ctrl
//PolyCopyingC<EditInt, Ctrl> a;
//CopyableC<Ctrl>* p = a.Copy();
//p->GetC().SizePos();

template<class T, class C>
class PolyCopyingC : public PolyDeepCopyNew<PolyCopyingC<T,C>, T>, public CopyableC<C>
{
public:
 virtual PolyCopyingC* Copy() const { return new PolyCopyingC(); }
 virtual const C& GetC() const { return *this; }
 virtual C& GetC() { return *this; }
};
#endif

///

 PolyCopyingC<EditInt, Ctrl> abc; //provide common base class info

 CopyableC<Ctrl>* pabc = abc.Copy();
 CopyableC<Ctrl>* pabc2 = pabc->Copy();

 pabc->GetC().SizePos();
 Add(*pabc);
 Add(pabc2->GetC().HSizePos().TopPos(0,100));
```