Subject: Re: Core chat...
Posted by mirek on Fri, 26 Oct 2007 12:01:33 GMT
View Forum Message <> Reply to Message

mdelfede wrote on Fri, 26 October 2007 07:03luzr wrote on Fri, 26 October 2007 12:42
Sorry. My mistake. Forget about [10]. Only

ctrls.Create<Button>();

or (to be more clear):

Array<Ctrl> a, b;
a.Create<Label>();
a.Create<EditField>();

b = a;

b.Create<Label>(); // Now what?!

Array<Ctrl> a, b; // two empty arrays, reference to nothing
a.Create<Label>(); // adds a Label to a, a becomes an array with a single reference at underlying data
a.Create<EditField>(); // adds an EditField to a, a is grown by 1 element, as it had a single reference to data, nothing strange happens.
b = a; // now a AND b have both reference to the same underlying array
b.Create<Label>(); // underlying data is copied ....

How is it copied? There is no copy operation for widgets. And it does not even have sense.

Quote:
[/code]
I see your point, here you *did* want a single array of controls, and you *did* want to destroy a when copying to b. You could have easily write :

Array<Ctrl> a;
a.Create<Label>();
a.Create<EditField>();

Array<Ctrl>&b = a;

b.Create<Label>();

That leaves both a and b pointing to SAME array.

Of course, but that is completely different thing.

In fact, yes, one of reasons to give away with reference counting for containers (or other "smart" approaches) is that you only seldem need to transfer the value of container at all.

OTOH, if you DO need such operation, then in most cases you either need exactly "pick" behaviour or you just do not care (function return value). Very seldom you need "deep copy" - in that case, you still have optional deep operations available.

And yes, sometimes you get "broken pick semantics" runtime assert. But I get 100 times more "invalid index" asserts than this one.

Quote:
You then could say that when a is destroyed, b points to nothing, that's true, but I can hardly imagine such a case. For example :

```
Array<Ctrl> MyFunc()
{
  Array<Ctrl> a;
  a.Create<Label>();
  a.Create<EditField>();

  Array<Ctrl>&b = a;

  b.Create<Label>();

  return b;
}

Array<Ctrl> c = MyFunc();
```

still works...

But does not compile Your template needs deep public copy constructor for T, even if it is not actually used (because reference count is always 1 for COW ops).

Quote:
and I can tell you that if you forget the & you have double controls on screen!

Sounds like a possible solution, but in that case you need to solve the quite complex problem of polymorphic copy - and all that only to make flawed code working somehow.