Subject: NTL - "deep copy semantics"?
Posted by Werner on Fri, 21 Jul 2006 15:57:51 GMT
View Forum Message <> Reply to Message

When reading about "deep copy" I feel a little bit confused.

You call it "deep copy semantics" to put a value into the target while preserving it in the source. And you set this in contrast to "pick transfer semantics" where a value is put into the target while destroying it in the source.

When I compare this with other C++ reading stuff I understand your "deep copy" as a simple copy process while "pick transfer semantics" means the same as "destructive copy semantics". The latter choice of words reserves the expression "deep copy" (vs "shallow" (or "flat") copy) to tell apart memberwise copy from bitwise copy.

Of course you have every right to define expressions to your liking as long as you make clear what you mean (and you do this!). But what is the reason for this non-customary use of these expressions? Or do I misinterpret something?

Werner

Subject: Re: NTL - "deep copy semantics"?
Posted by mirek on Sun, 23 Jul 2006 17:09:15 GMT
View Forum Message <> Reply to Message

Actually, the existing terminology is not stable and e.g. "destructive copy semantics" can mean more than single thing.

Especially, definitions vary about what happens to source instance. Most "destructive copy" articles I have read simple reset it to some form of "empty" value.

Sometimes also the term "destructive copy" is used to actually describe "move" (as with NTL "Moveable").

Anyway, the important aspect of "pick" that makes it different from closest definition of "destructive copy" is the fact that it puts source to specific "picked" state (rather than simply emptying it). That in fact is very helpful to catch many associated bugs.

I have also decided that "transfer" is better term that "copy", because "copy" at least to me implies that the source is unchanged... (ok, perhaps stretching it too far

Mirek

Subject: Re: NTL - "deep copy semantics"?
Posted by nixnixnix on Thu, 23 Aug 2007 02:17:38 GMT

Hi there,

Please ignore this - I found the DeepCopy thing confusing so I leave this here for those who come after me

I don't see any examples of DeepCopy but if I understand correctly, my class, the declaration for which looks like this

```
class Mine
{
public:
  Mine();

  double fMember;
  int nMember;

  Array <Point> m_pts;
};
```

can be copied with the source in tact so long as I write it

```
class Mine
{
public:
  Mine();

  double fMember;
  int nMember;

  WithDeepCopy<Array <Point> > m_pts;
};
```

and this means that I can use the default copy constructors and forget about making my own unless I really want to. When I call

```
Mine mine;
// some initialisation
..
Mine yours(mine);
```

yours copies mine and they both have all the same members including in their arrays.

Nick

Subject: Re: NTL - "deep copy semantics"?
Posted by mirek on Thu, 23 Aug 2007 15:45:10 GMT
View Forum Message <> Reply to Message

Well, if you use WithDeepCopy here, you will get "deep copy" default copy constructor.

Without "WithDeepCopy", you would get default "pick" constructor.

Mirek

Subject: Re: NTL - "deep copy semantics"?
Posted by nixnixnix on Wed, 05 Sep 2007 02:43:07 GMT
View Forum Message <> Reply to Message

Thanks Mirek - that all works fine.

I have another related question: How exactly does Array support polymorphism? In the documentation it says that you pass a pointer but this seems meaningless as all data structures are polymorphic if you use them to store pointers as a pointer can point to anything. In any case I tried this and then copied my object and found that the pointer values are copied but not the objects they point to. A simplified analogue of my problem is below...

```
class VPoint
{
  Pointf m_pt;
  ...
  ...
}

class PointLayer
{
  WithDeepCopy <Array <VPoint> > m_pts;
  ...
  .../* lots of functions for using and drawing and editing VPoints */
}

class House : VPoint
{
  double m_fNoise;
  ...
}

class HouseLayer : PointLayer
{
```

```
  ...
}
```

Now I want my HouseLayer to contain an array of House objects in place of the VPoint objects in its base class but to use all the functionality that I've written into PointLayer but using House objects rather than VPoint objects.

I thought that the polymorphism in Array would let me do that but it appears not. I don't appear to be able to store House objects in the PointLayer::m_pts array and if I store pointers in my Array then my House objects and VPoint objects don't get copied.

I don't know if this is a UPP question or just to do with my lack of knowledge of C++. Any pointers (excuse the pun) would be really appreciated though.

Been searching through Stroustrup's book but not finding anything useful. I know I can do what I want to do if I just split my House object into the VPoint part and the House part and have another Array in my HouseLayer object and then keep the two arrays in sync but I have this overwhelming feeling that C++ is not that messy and that a more elegant solution exists.

Subject: Re: NTL - "deep copy semantics"?
Posted by mirek on Wed, 05 Sep 2007 06:45:32 GMT
View Forum Message <> Reply to Message

This is quite hard to answer... This does not seem a problem of inheratince only, but also the problem of relations of "Layer" and element classes.

Inheritance in OOP languages is about single class only, not about a couple of classes. If there is more than one, you have to do something more.

What exactly and how depends on particular problem.

Subject: Re: NTL - "deep copy semantics"?
Posted by mr_ped on Thu, 06 Sep 2007 01:05:05 GMT
View Forum Message <> Reply to Message

The memory footprint of class VPoint and House is different, which pretty much breaks any chance of direct "polymorphism" in Array.

Pointers to VPoint can be set to point to House too, and the HouseLayer can typecast the pointers back to House pointers in case it is sure there's House and not VPoint upon that pointer.
(or to do checks of type during pointer casting with RTTI)

If you want to keep the DeepCopy behaviour too (i.e. not just to copy pointer value, but actually to

create new instance of VPoint or House, copy the data, and set up the pointer to new instance in the copy of PointLayer), you will need to wrap it up in new class like:
class PointContainer {
  VPoint *ptr;

  ...here comes copy constructor which will create new instance of VPoint or House, copy data from old pointer, and set up the new pointer to new copy of PointContainer object...
}

This is IMHO very cumbersome in C++ and some overall class design should be rather changed, than trying to do it this way.

I think using pointers will be probably the right answer for this, but I think you should avoid copy behavior, and suffice with pointers copying only, which will point to shared instance of data. These things are prone to memory leaks, so be careful with memory releasing or use smart pointers / Garbage collector.

Still the PointLayer will work only with VPoint pointers in functions, whenever you will want to work with House instance, you will need to cast the pointer, and that should be done probably only by HouseLayer function, i.e. you can either write all those functions twice with minor changes, or use templates...

Anyway, your PointLayer vs HouseLayer class suggest your class design is too much data oriented, not enough abstract probably.

If House is on same level of abstraction than VPoint (and your usage in Layers suggests so), it should share the basic functions interface with with it (functions for layers to manipulate with them).

I.e.

```
class VPoint {
 virtual MoveMe(...);
 virtual HideMe(...);
};

class House : public VPoint {
 aditional data;
 ..custom version of MoveMe() to handle my additional data;
};

class Tree : public VPoint {
 different additional data;
 ..custom version of MoveMe();
};

class Layer {
 array of VPoint pointers;
```

```
 void MoveAllMembers() {
   for each in pointers do
     pointers->MoveMe(); //I don't care if you are House or Tree or VPoint, just move.
 }
}
```

Actually here the House and Tree is on same level of abstraction, VPoint is base class and shouldn't be used directly too much probably. (Unless it feels right, like in Layer way)

If you want to do something specific upon House, you should either have different list of House objects only, or to have a way to identify House pointers within Layer, typecast them back on House pointer, and call special House functions...

All this should be outside of Layer class, which is not supposed to make difference between VPoint and House or Tree.

I think I can't give you better advice without having better idea what you are trying to do.. and even then I will probably not help you too much, I never really designed any serious OOP application, so I lack experience, I'm more a theory guy..

Subject: Re: NTL - "deep copy semantics"?
Posted by nixnixnix on Sun, 09 Sep 2007 00:36:51 GMT
View Forum Message <> Reply to Message

Thanks Mirek,

The fact that you think that what I was thinking was possible was very hard if at all possible, reassured me that I was barking up the wrong tree

Thanks Mr Ped,

I can't share instances in this case as what I need to do is to create independent objects which can share base functionality (I do share instances elsewhere in the same app). However, most of what you said is spot on. I can add additional data to the derived class and still use the base functionality. I just need to accept that the derived class has to manage its specific data which is obvious really when you think about it. I think I was getting just a bit too carried away with the awesome power and beauty of OOP.

Now I've implemented what I originally thought was the "messy" way I see that it is still incredibly elegant and that my derived classes only need to handle their own data and can override the base functionality or not as desired. The only "mess" is two arrays to handle what are in effect the same objects and even then there is only one array explicitly declared in the base class and then the extra array specified in the derived class to hold the additional data.

Thanks to both of you for answering and for letting me know when I'm starting to run up against the edges of the ocean of OOP rather than just my usual small sand-bank in the fog

Nick