Subject: Questions about static casting Polymorphic Array Elements, iterator, Ptr and Pte
Posted by navi on Mon, 24 Dec 2012 00:58:02 GMT
View Forum Message <> Reply to Message

```
struct shape{
 int type;
}

struct circle : shape{
 circle(){ type=1; }
 int radius;
 int x, y;
}

struct triangle : shape{
 triangle(){ type=2; }
 int x[3], y[3];
}

struct rectangle : shape{
 rectangle(){ type=3; }
 int x[4],y[4];
}

Array<shape> a;

a.Add(new circle);
a.Add(new triangle);
a.Add(new rectangle);

if(a[1].type==2){

 triangle *m = static_cast<triangle *> (&a[1]);
}
```

in the above example, is this the correct Syntax & correct way to static casting Polymorphic Array Elements?triangle *m = static_cast<triangle *> (&a[1]);

How do I create an iterator and static cast the iterator instead? Is this a better way then the previous method?

What is the correct way to static cast an Element from Polymorphic Array?

As suggested in NTL Tutorial - Point.6 Polymorpic Array is great for the purpose of storing Polymorphic objects. no problem of object slicing. No need for memory management. So, the

below question is purely for the curiosity sake.

Are there smart pointers in U++? "Stack Overflow" and other forums people suggesting to use STL vector of smart pointers to keep Polymorphic Elements. Is there any way to achieve that using U++ NTL? I have seen Ptr and Pte in the Manuel but confuse about what do they actually do? do they only assign null when object is destroyed or do they actually destroy pointing object (i.e. manages the object De-Allocation?) when they go out of scope?

Thanks & Regards
Navi

Subject: Re: Questions about static casting Polymorphic Array Elements, iterator, Ptr and Pte
Posted by mirek on Mon, 24 Dec 2012 08:50:26 GMT
View Forum Message <> Reply to Message

navi wrote on Sun, 23 December 2012 19:58struct shape{
 int type;
}

struct circle : shape{
 circle(){ type=1; }
 int radius;
 int x, y;
}

struct triangle : shape{
 triangle(){ type=2; }
 int x[3], y[3];
}

struct rectangle : shape{
 rectangle(){ type=3; }
 int x[4],y[4];
}

Array<shape> a;

a.Add(new circle);
a.Add(new triangle);
a.Add(new rectangle);

Event better (more "U++ish") is to use

circle& c = a.Create<circle>();

here.

Quote:
in the above example, is this the correct Syntax & correct way to static casting Polymorphic Array Elements?triangle *m = static_cast<triangle *> (&a[1]);

Yep. Although, IME, if you use the trick above, you in fact seldom need to cast later. I guess that you only need to know the final type to initialize it, rest can be taken care about with virtual methods.

Quote:
How do I create an iterator and static cast the iterator instead? Is this a better way then the previous method?

Well, U++ way is to prefer index based iteration over iterator. Anyway, if you insist, you can of course use iterator.

```
#include <Core/Core.h>

using namespace Upp;

struct Foo {
 int foo;
};

struct Bar : Foo {
 int bar;
};

CONSOLE_APP_MAIN
{
 Array<Foo> x;
 x.Create<Bar>().bar = 54321;

 for(Array<Foo>::Iterator it = x.Begin(); it != x.End(); ++it)
  DUMP(static_cast<Bar &>(*it).bar);
}
```

Quote:
Are there smart pointers in U++?

Well, there are some historical in 'non-canonical' parts of U++ that are not normally part of U++ releases, anyway, shared smart pointers are generally considered "BIG EVIL", something to avoid.

Quote:
I have seen Ptr and Pte in the Manuel but confuse about what do they actually do? do they only assign null when object is destroyed

Yes. Thing is, the U++ way nicely solves most issues about resource management, so that you never call 'delete' in high-level code, but the solution has somewhat weak spot that you have to be careful about dangling pointers. In some cases, Ptr is a good tool to deal with them...

Quote:
or do they actually destroy pointing object (i.e. manages the object De-Allocation?) when they go out of scope?

No.

Mirek

Subject: Re: Questions about static casting Polymorphic Array Elements, iterator, Ptr and Pte
Posted by navi on Mon, 24 Dec 2012 09:57:56 GMT
View Forum Message <> Reply to Message

Dear Mirek,

First of all, Thank you for the quick response.

mirek wrote on Mon, 24 December 2012 09:50
Event better (more "U++ish") is to use

circle& c = a.Create<circle>();

here.

Quote:
in the above example, is this the correct Syntax & correct way to static casting Polymorphic Array Elements?triangle *m = static_cast<triangle *> (&a[1]);

Yep. Although, IME, if you use the trick above, you in fact seldom need to cast later. I guess that you only need to know the final type to initialize it, rest can be taken care about with virtual methods.

I will use the U++ method Create<type>() when I first create and add element into the array.
I am casting element at a later time depending on the "type" variable which can be access via the base type pointer/referance returned by the Array's index operator, to access the derived part of the object. I am under impression that since the Array is of the base type, I cant access the derived part of the object using the operator[] without casting. As it might return a reference of the base type. does Array return a derived type reference if added using Create<type>()? does Array actually remembers type for individual elements?! if so, then its awesome! I don't need casting what so ever at later time.

P.S. I do not have any virtual method in my objects other then the virtual destructor. which is also empty at the point. I mainly need to retrieve and set various public variables exposed by the different kind of objects. for instance circle only has 3 separate int variable, where triangle has an int array of size 3. some objects might even have other different type of variable like String, Date and etc.

mirek wrote on Mon, 24 December 2012 09:50...anyway, shared smart pointers are generally considered "BIG EVIL", something to avoid....

...Yes. Thing is, the U++ way nicely solves most issues about resource management, so that you never call 'delete' in high-level code, but the solution has somewhat weak spot that you have to be careful about dangling pointers. In some cases, Ptr is a good tool to deal with them...

Understood. pte/ptr is NOT Equal to "shared smart pointers"

Thanks & Regards
Navi

Subject: Re: Questions about static casting Polymorphic Array Elements, iterator, Ptr and Pte
Posted by mirek on Mon, 24 Dec 2012 13:40:05 GMT
View Forum Message <> Reply to Message

navi wrote on Mon, 24 December 2012 04:57
I am casting element at a later time depending on the "type" variable which can be access via the base type pointer/referance returned by the Array's index operator, to access the derived part of the object. I am under impression that since the Array is of the base type, I cant access the derived part of the object using the operator[] without casting.

Correct, but virtual methods of that type CAN.

Of course, it is not an universal truth and sometimes casting you suggest is really needed, but quite often you can better handle issues by good design of base class interface, with the right set of virtual methods doing things.

All in all, the whole purpose of such design is to handle all subclasses the same way...

To illustrate, say that your classes are intended for some graphical designer and that you want a menu with a set of operations for each type. Then you can do something like

```
if(a[1].type==2){
 triangle *m = static_cast<triangle *> (&a[1]);
       CreateMenuForTriangle(m);
}
```

or you can just use

```
struct shape {
   virtual void Menu(Bar& bar);
}
```

Mirek

Subject: Re: Questions about static casting Polymorphic Array Elements, iterator, Ptr and Pte
Posted by navi on Mon, 24 Dec 2012 20:39:18 GMT
View Forum Message <> Reply to Message

Quote:... Correct, but virtual methods of that type CAN.

Of course, it is not an universal truth and sometimes casting you suggest is really needed, but quite often you can better handle issues by good design of base class interface, with the right set of virtual methods doing things.

All in all, the whole purpose of such design is to handle all subclasses the same way...

I think I understand now what you meant by "...rest can be taken care about with virtual methods.". I you design your classes well then it is possible to just getaway with casting once. and rest can be done by calling virtual methods that are accessible through base type pointer/reference. Thanks for explaining it to me in that detail!

Regards
Navi