

# When Nothing Matters

## (or the story of the dangling pointer)

by Joanna Carter

From time to time, someone will come on the newsgroups complaining that nilling a reference to an object does not nil all references to that object. They have encountered the problems caused by what is known as a dangling pointer.

Now, the design techniques employed by programmers who end up with dangling pointers are open to question. I thought it would be interesting, nonetheless, to discuss one possible solution to resolve some of the problems caused when an attempt is made to free an object in such a way that all references to that object could indicate the demise of that object by exhibiting a nil value.

For our example, we will take a totally meaningless class 'TThing' and modify it to allow us the facility to create instances and to assign more than one reference to an instance, but that calling the Free method on any one of those instances would ensure that all references to the instance were automatically set to nil.

---

```
TThing = class
private
  class procedure NilReferences(var obj: TThing);
public
  class function NewInstance: TObject; override;
  procedure FreeInstance; override;
end;
```

---

Ideally, I would have liked to be able simply to override the NewInstance and FreeInstance methods and add the code that added a reference to the new object or nilled all the references to these methods. But unfortunately the address of the result of any function is not the same as the address of the variable used to hold the returned instance and it is the addresses of the variables that will hold instances that we are after.

We also have a problem in that we are unable to override the assignment operator that would be used to assign the instance held by one variable to another.

So, instead of using the normal constructor of a class to create instances and using the ':=' operator to assign instances, we have to use a couple of class methods to fulfil these behaviours, but we can still override the FreeInstance method to call our private method that will be responsible for nilling our references.

---

```
TThing = class
private
  class procedure NilReferences(var obj: TThing);
public
  class procedure CreateInstance(var obj: TThing);
  class procedure AddReference(var obj, newObj: TThing);
  procedure FreeInstance; override;
end;
```

---

The first thing we need to add to the implementation section of our class's unit is the list that is going to hold on to the addresses of our variables:

---

```
implementation

var
  _ObjectRefs: TList;
```

---

The class method that we will have to substitute for a constructor is simplicity itself, as is the class method that we will need to call instead of using the assignment operator.

---

```

class procedure TThing.CreateInstance(var obj: TThing);
begin
  obj := Create;
  _ObjectRefs.Add(@obj);
end;

class procedure TThing.AddReference(var obj, newObj: TThing);
begin
  newObj := obj;
  _ObjectRefs.Add(@newObj);
end;

```

---

All that is happening in these methods is that, whether we are creating or assigning, the address of the variable being assigned to is added to our internal list.

Now, when it comes to freeing an instance and nilling all those references, I have come up with one possible way of solving the problem; you may well find others and it would be interesting to see what other solutions are out there. I must stress that this code is more than likely not thread-safe and that I still cannot account for objects added to TObjectLists.

---

```

class procedure TThing.NilReferences(var obj: TThing);
var
  i: Integer;
  remList: TList;
begin
  remList := TList.Create;
  try
    for i := 0 to _ObjectRefs.Count - 1 do
      if TThing(_ObjectRefs[i]^) = obj then
        begin
          remList.Add(Pointer(i));
        end;

    for i := remList.Count - 1 downto 0 do
      begin
        Pointer(_ObjectRefs[i]^) := nil;
        _ObjectRefs.Delete(Integer(remList[i]));
      end;
    finally
      remList.Free;
    end;
  end;
end;

```

---

I use a two-pass routine; the first pass goes through the list of addresses, extracting all those references that pertain to the instance that we wish to free and nil, the second goes through the extracted list of references and nills them.

---

```

procedure TThing.FreeInstance;
begin
  NilReferences(self);
  inherited FreeInstance;
end;

```

---

All that is left to do in this class is to override the FreeInstance method to call our NilReferences method before going on to free the instance itself.

---

```

var
  Thing1: TThing;
  Thing2: TThing;
  Thing3: TThing;
begin

```

```
TThing.CreateInstance(Thing1);  
TThing.AddReference(Thing1, Thing2);  
TThing.AddReference(Thing2, Thing3);  
Thing2.Free;  
  
end;
```

---

The test code is somewhat unconventional in that it doesn't obviously use the constructor or assignment operator, but if you run this through the debugger and add watches on the two variables, you will see that calling Free on any of the variables sets all the variables to nil.

## Conclusion

Whether this is a truly useful technique or not has to be in the mind of the reader. It certainly solves a problem, but whether that problem is common or great enough to warrant the effort expended in creating the solution, is something that only those who are plagued with an attack of the dangling pointers can say.

Nevertheless, I have demonstrated that it is possible to set as many references to a single object to nil simply by calling the destructor on an instance. So you see, nothing really can matter.



Joanna is a Consultant Software Engineer whose work can take her anywhere in the world. Her work involves architecting systems and teaching OO principles and practice to developers. She is also a member of Borland's TeamB (the equivalent of Microsoft's MVPs), a group of technical experts who monitor and advise on the Borland newsgroups. At present Joanna is experimenting with porting OO frameworks originally written in Delphi to .NET and in researching the use of generic coding techniques in .NET 2.0.

---