# Reducing code with scoped objects

Michaël Van Canneyt

May 18, 2025

**Abstract**

With the ability to create custom managed records, it is now possible to reduce the number of `try..finally` blocks in your code.

## 1 Introduction

Pascal is not a garbage collected language. That means that you are responsible for memory management. Any resource (or memory) that you allocate, must also be freed. Avoiding memory leaks can take some work.

Luckily, there are some ways to reduce the work that this manual memory management requires:

- Using interfaces: an interface is reference counted. When the last reference to an interface goes out of scope, the underlying object is freed.
- Using dynamic arrays: similar to interfaces, dynamic arrays are reference counted. When the last reference to the dynamic array goes out of scope, the memory occupied by the array. If the array contains objects, the objects are *not* freed, however.
- Strings (unicode and ansi strings) are also reference counted, similar to dynamic arrays.
- Using `TComponent` to construct trees of objects. A parent component owns all its children: when the parent is freed, automatically all children are freed. This is traditionally how Delphi cleans up forms and datamodules.
- For lists of objects, the `TObjectList` and `TDictionary` can be told that they own the objects contained in them, and they will free them when the list or dictionary is freed.

If the object you are using is not one of the above, then you are responsible for freeing the object. This means that you'll typically encounter code like the following:

```
Procedure DoSomeWork;
var
  ListA,ListB : TStrings;
begin
  ListA:=TStringList.Create;
  try
    // Do something with ListA
    ListB:=TstringList.Create;
```

```
    try
      // Do something with ListA and ListB
    finally
      ListB.Free;
    end;
  finally
    ListA.Free;
  end;
end;
```

The experienced programmer will reduce this to the following:

```
Procedure DoSomeWork;
var
  ListA,ListB : TStrings;
begin
  ListB:=nil;
  ListA:=TStringList.Create;
  try
    // Do something with ListA
    ListB:=TstringList.Create;
    // Do something with ListA and ListB
  finally
    ListB.Free;
    ListA.Free;
  end;
end;
```

Because `Free` checks whether the object is not `Nil` before calling `Destroy`, this is perfectly safe in case an exception happens before B is created.

Is there a way to remove these `try..finally` blocks from the code ? Fortunately, there is: management operators.

## 2 Management operators

Both Delphi and Free Pascal have added management operators to the list of operators that can be defined on a record type. Interfaces, dynamic arrays and strings are managed types: this means that Delphi under the hood adds code to manage their lifecycle: they are initialized empty when they come into scope, and they are finalized when they go out of scope. When copying, the reference counts are adjusted. This is also done for arrays of these types or records that contain a managed type.

Management operators allow one to perform this task implicitly on records. The definition of a management operator is as follows in Delphi:

```
Type
  TMyRecord = record
    Obj: TMyObject;
    class operator Initialize (out Dest: TMyRecord);
    class operator Finalize (var Dest: TMyRecord);
  end;
```

the `Obj` field can be anything, but we choose this for a reason which will become apparent soon. The `Initialize` operator is called when a variable of type `TMyRecord` comes into scope, passing it the new variable in `Dest`. The `Finalize` operator will be called called when the variable goes out of scope, again passing it the variable that goes out of scope.

In Free Pascal, the definition is slightly different:

```
Type
  TMyRecord = record
    Obj: TMyObject;
    class operator Initialize (var Dest: TMyRecord);
    class operator Finalize (var Dest: TMyRecord);
  end;
```

The `out` parameter for the initialize record must be a `var` parameter instead.

One way to use this would be to create and destroy the object:

```
class operator TMyRecord.Initialize (out Dest: TMyRecord);
begin
  Dest.Obj:=TMyObject.Create;
end;

class operator TMyRecord.Finalize (var Dest: TMyRecord);
begin
  FreeAndNil(Dest.Obj);
end;
```

This means that the following will work:

```
procedure DoTest;
var
  Rec : TMyRecord;
begin
  Writeln(Rec.obj.classname)
end;
```

When the procedure is called, Delphi or Free Pascal will call the `Initialize` operator on `Rec`, which will create the `TMyObject` instance and store it in the `Obj` field. When the procedure exits, it will call the `Free` method.

The nice thing about this is that the compiler will create an implicit `try..finally` block around the procedure body, and calls the `Finalize` operator in the `finally` block, just as it does this for interfaces, strings and dynamic arrays: the record will always be properly finalized, without the need for try..finally blocks.

# 3 Introducing `TScoped`

With the above management operators, we have done away with the try finally block and the need to explicitly free the object `TMyObject` whenever it is used. Of course, it would not be practical to have to define a record for every object type that is about to be used in a program. Luckily, this is not necessary. Using generics, we can create a record definition that handles any kind of `TObject`.

In the development version of Free Pascal, `TScoped` has been defined as follows:

```
generic TScoped<T:class> = record
private
  obj: T;
public
  class operator Initialize(var hdl: TScoped);
  class operator Finalize(var hdl: TScoped);
  class operator :=(aObj : T) : TScoped;
  class operator :=(const aObj : TScoped) : T;
  procedure Assign(aObj : T); inline;
  function Swap(AObj: T): T;
  function Get : T;
end;
```

The implementation of this record is quite simple:

```
class operator TScoped.Initialize(var hdl: TScoped);
begin
  hdl.obj := nil;
end;

class operator TScoped.Finalize(var hdl: TScoped);
begin
  hdl.obj.free;
  hdl.obj:=nil;
end;

procedure TScoped.Assign(aObj : T);
begin
  Self.Obj:=aObj;
end;

function TScoped.Swap(AObj:T):T;
var
  LCurrent:T;
begin
  LCurrent := self.obj;
  Assign(AObj);
  Result := LCurrent;
end;

function TScoped.Get() : T;
begin
  Result := self.obj;
end;

class operator TScoped.:=(aObj : T) : TScoped;

begin
  result.assign(aObj);
end;

class operator TScoped.:=(const aObj : TScoped) : T;

begin
```

```
  Result:=aObj.Get();
end;
```

In Delphi, the definition looks as follows:

```
  TScoped<T:class> = record
  Private
    obj: T;
  Public
    class operator Initialize(out hdl: TScoped<T>);
    class operator Finalize(var hdl: TScoped<T>);
    class operator Implicit(const aObj : TScoped<T>) : T;
    class operator Implicit(const aObj : T) : TScoped<T>;
    class operator Assign (var Dest: TScoped<T>; const [ref] Src: TScoped<T>);
    procedure Assign(AObj: T);
    function Swap(AObj: T): T;
    function Get() : T;
  end;
```

Using this generic record, it means we can now do the following:

```
Procedure DoSomeWork;
var
  ListA,ListB : TScoped<TStringList>;
begin
  ListA.Assign(TStringList.Create);
  ListB.Assign(TStringList.Create);
  // Do something with ListA and ListB
end;
```

Since we overloaded the assignment operator (:= or implicit), we can also write the following:

```
Procedure DoSomeWork;
var
  ListA,ListB : TScoped<TStringList>;
  I : Integer;
begin
  ListA:=TStringList.Create;
  ListB:=TStringList.Create;
  for I:=ListA.Get.Count-1 downto 0 do
    ListB.Get.Add(ListA.Get[i])
end;
```

Note that to get to the actual object, you need to use the Get function. This can get quite cumbersome, so one can do this:

```
Procedure DoSomeWork;
var
  ListA,ListB : TScoped<TStringList>;
  A,B : TStringList;
  I : Integer;
begin
  ListA.Assign(TStringList.Create);
```

```
  A:=ListA;
  ListB:=TStringList.Create;
  B:=ListB;
  for I:=A.Count-1 downto 0 do
    B.Add(A[i]);
end;
```

It is tempting to add the `Explicit` operator to the `TScoped` definition, and allow the object to be typecasted to the `TScoped` record:

```
class operator TScoped<T>Explicit(const aObj : T) : TScoped<T>;
begin
  Result.Assign(aObj);
end;
```

This would allow to write the following:

```
Procedure DoSomeWork;
var
  A,B : TStringList;
  I : Integer;
begin
  // typecast !
  A:=TScoped<TStringList>(TStringList.Create);
  B:=TScoped<TStringList>(TStringList.Create);
  for I:=A.Count-1 downto 0 do
    B.Add(A[i]);
end;
```

However, this would lead to errors. To understand why, let us examine what happens when we write this:

```
  A:=TScoped<TStringList>(TStringList.Create);
```

For the right-hand side of the assignment, the compiler will create a temporary variable, typecast the created stringlist to it, and then uses the `Implicit` operator to assign it to `A`. In effect, something akin to the following would be executed (assume the variable is called `tmp`):

```
var
  Tmp: TScoped<TStringList>;
begin
  // Translation of explicit typecast operator
  Tmp:=TStringList.Create;
  // Use of implicit operator to assign to A
  A:=Tmp;
```

All seems to be in order, except that the lifetime or scope of this temporary variable (`tmp`) is not well-defined: the compiler may decide to use it to assign B as well, or may decide to reuse it later on. In that case, it will finalize the `tmp` variable before the end of the procedure, and `A` would be left pointing to a non-existent instance of `TStringList`.

It is also possible to use the `TScoped` definition in classes. See the following example:

```
  TChild = class
  private
    FStr : String;
  public
    class var InstanceCount : Integer;
  public
    constructor Create(const AStr:String);
    destructor Destroy(); override;
    procedure Display();
  end;

//type
  TParent = class
  private
    FStr : String;
    ScopedChild: specialize TScoped<TChild>;
  public
    class var InstanceCount : Integer;
  public
    constructor Create(const AStr:String);
    destructor Destroy(); override;
    procedure Display();
  end;
```

With the following implementation for the child:

```
constructor TChild.Create(const AStr:String);
begin
  FStr := AStr;
  Inc(InstanceCount);
end;

destructor TChild.Destroy();
begin
  Dec(InstanceCount);
end;

procedure TChild.Display();
begin
  writeln('ChildStr = ' + FStr);
end;
```

And the following implementation for the parent class:

```
constructor TParent.Create(const AStr:String);
begin
  FStr := AStr;
  inc(InstanceCount);
  ScopedChild.Assign(TChild.Create('"ChildStr - ' + AStr+'"'));
end;

destructor TParent.Destroy();
begin
  Dec(InstanceCount);
```

```
end;

procedure TParent.Display();
begin
  ScopedChild.Get.Display();
end;
```

Note that the `TParent` destructor does not destroy the child instance. This is done automatically because when the parent is freed, the `ScopedChild` field goes out of scope and is finalized.

We can test this in the following routine:

```
procedure DoTest;
var
  P: TScoped<TParent>;
begin
  P.Assign(TParent.Create('Hello'));
  P.Get().Display();
end;

begin
  DoTest;
  Writeln('Counts: ',TParent.InstanceCount,'-',TChild.InstanceCount);
end;
```

This will display a count of zero for both objects: all objects are freed automatically without the need to have lots of `try..finally` blocks or `Free` statements.

Under usual circumstances, a `TScoped` is not copied: it just serves as a container for another class. However, there may be circumstances in which you want to copy a `TScoped`. In Free Pascal, do not assign one TScoped to another by direct assignment:

```
Procedure DoSomeWork;
var
  A,B : TScoped<TStringList>;
begin
  A.Assign(TStringList.Create);
  B:=A;
end;
```

It will compile, but in this case, both `A` and `B` will hold a reference to the stringlist, and will free them both at the end of the procedure, resulting in a double free. In Delphi it is OK to do so, because delphi allows to override the assign operator for two `TScoped` variables.

If you really need to copy the `TScoped`, this is the correct way to do it in Free Pascal:

```
Procedure DoSomeWork;
var
  A,B : TScoped<TStringList>;
begin
  A.Assign(TStringList.Create);
  B.Assign(A.Swap(nil);
end;
```

# 4 Locking

When making multithreaded applications, it is common to protect access to shared resources with a `TMutex` or `TCriticalSection`: these synchronization objects allow to control access to resources that are shared between various threads.

This means you'll see lots of code like the following:

```
procedure TMyObject.ChangeSomeResource;

begin
  FLock.Enter;
  try
    // Do some things.
  finally
    FLock.Leave;
  end;
end;
```

Only one thread at a time can execute the code between the `try..finally` block. This block is needed to make sure that in case of exception, the lock (or mutex) is released.

The `TScoped` cannot be used for this:

```
procedure TMyObject.ChangeSomeResource;
var
  lLock:TScoped<TCriticalSection>;
begin
  lLock.Assign(Flock);
  lLock.Get.Enter;
  // Do some things.
end;
```

At the end, the lock would be destroyed, instead of simply unlocked.

Fortunately, we don't have to actually destroy the lock in the finalization of our record. We can simply release the lock. A record that does just that has been implemented in the `syncobjs` unit of free pascal:

```
generic TLockGuard<T:TSynchroObject> = record
  obj: T;
  class operator Initialize(var hdl: TLockGuard);
  class operator Finalize(var hdl: TLockGuard);
  procedure Init(AObj: T);
end;
```

Its implementation is quite simple:

```
class operator TLockGuard.Initialize(var hdl: TLockGuard);
begin
  hdl.obj := nil;
end;

class operator TLockGuard.Finalize(var hdl: TLockGuard);
begin
```

```
  if (hdl.obj=nil) then
    exit;
  hdl.obj.Release();
end;

procedure TLockGuard.Init(AObj:T);
begin
  self.obj := AObj;
  self.obj.Acquire();
end;
```

As you can see, in the finalization call, we simply call the `Release` method of the synchronization object.

So now we can do without an explicit `try..finally` block and write:

```
procedure TMyObject.ChangeSomeResource;
var
  lLock:TLockGuard<TCriticalSection>;
begin
  lLock.Init(Flock);
  // Do some things.
end;
```

To demonstrate this, we'll create a small demo program that demonstrates this. We'll create a thread that calculates a fibonacci number, and we'll protect the calculation using a lock, so only one thread at the time performs the calculation.

Here is the fibonacci routine:

```
Function Fibonacci(TN,N : Integer) : Int64;
Var
  Next,Last : Int64;
  I : Integer;

begin
  if N=0 then
    exit(0);
  Result:=1;
  Last:=0;
  for I:=1 to N-1 do
    begin
    Next:=Result+last;
    Last:=Result;
    Result:=Next;
    Writeln('Thread['+IntToStr(TN)+'] '+IntToStr(Result));
    end;
end;
```

The `TN` parameter is there for display purposes only, to make it clear in the output which thread is performing the calculation.

We then define our thread object, together with some count objects:

```
Type
  TCalcThread = Class(TThread)
```

```
  Public
    class var
    ExecuteLock : TCriticalSection;
    ThreadCount : Integer;
    ExecuteCount : Integer;
  Private
    FNo : Integer;
  Public
    constructor create(aNo : Integer);
    destructor destroy; override;
    Procedure Execute; override;
  end;
```

Each thread gets a number, to be able to identifiy it. The operating system's thread id could be used for this, obviously, but a simple integer reads more easily. The `ExecuteLock` critical section will be used to synchronize the execution.

The constructor and destructor increment and decrement the thread count, and display a diagnostic message:

```
constructor TCalcThread.create(aNo : Integer);
begin
  Inherited Create(False);
  InterlockedIncrement(ThreadCount);
  FNo:=aNo;
  Writeln('Creating thread ',FNo);
  FreeOnTerminate:=True;
end;

destructor TCalcThread.destroy;
begin
  InterlockedDecrement(ThreadCount);
  Inherited;
end;
```

Note that we set the thread to free itself when it is finished executing.

Finally, in the execute method, we use our `TLockGuard` (the example below uses FPC's syntax for specializing a generic). For display purposes, at the beginning and end of the routine, we also do a check that no more than 1 thread is calculating the fibonacci number .

```
procedure TCalcThread.Execute;
var
  lock : specialize TLockGuard<TCriticalSection>;
  Res : Integer;
begin
  lock.Init(ExecuteLock);
  InterlockedIncrement(ExecuteCount);
  if ExecuteCount<>1 then
    Writeln('Error : multiple threads are executing (start)');
  Res:=Fibonacci(FNo,10);
  writeln('Thread['+IntTostr(FNo),'] Fibonacci(10) = '+IntToStr(Res));
  InterlockedDecrement(ExecuteCount);
  if ExecuteCount<>0 then
```

```
    Writeln('Error : multiple threads are executing (stop)');
end;
```

In the main program we initialize the lock and start a number of threads. After that we wait till the threads are all done:

```
var
  I : integer;
begin
  TCalcThread.ExecuteLock:=TCriticalSection.Create;
  for I:=1 to 10 do
    TCalcThread.Create(i);
  repeat
    sleep(10);
    CheckSynchronize;
  until (ThreadCount=0);
  TCalcThread.ExecuteLock.Free;
end.
```

When you run this program, you'll see from the output messages that only one thread at a time is calculating the fibonacci number. Naturally, this is not something you would normally do, as it defeats the purpose of running a calculation in a thread in the first place.

The lock must be created (and destroyed) outside the routines that use them. The following will not work for obvious reasons:

```
procedure TCalcThread.Execute;
var
  lock : specialize TLockGuard<TCriticalSection>;
  Res : Integer;
begin
  lock.Init(TCriticalSection.Create);
  Res:=Fibonacci(FNo,10);
  writeln('Thread['+IntTostr(FNo),'] Fibonacci(10) = '+IntToStr(Res));
end;
```

The `syncobjs` unit already specializes the `TLockGuard` generic with the 3 synchronization objects defined in the SyncObjs unit:

```
TCriticalSectionGuard = specialize TLockGuard<TCriticalSection>;
TSemaphoreGuard = specialize TLockGuard<TSemaphore>;
TMutexGuard = specialize TLockGuard<TMutex>;
```

Using these pre-defined types saves you some typing and makes compilation slightly faster.

# 5 conclusion

In this article we've shown how you can use the management operators to make your life easier by reducing the number of `try..finally` blocks that you must write. It's likely that at the same time it makes execution slightly faster: chances are that your code already causes the compiler to insert an invisible `try..finally`

block, for instance when using ansistrings or unicodestrings. If so, the compiler can simply reuse this block to handle the finalization of the records. The code for the `TLockGuard` and `TScoped` generics is integrated in Free Pascal. For Delphi users, units with their definitions are includes with the code for this article. No doubt, more such techniques can be conceived, and the 2 records introduced here can serve as inspiration. The author wishes to thank Loïc Touraine for his initial implementation of these 2 useful records.