Stack-Based Objects In Delphi

Improving object construction and destruction

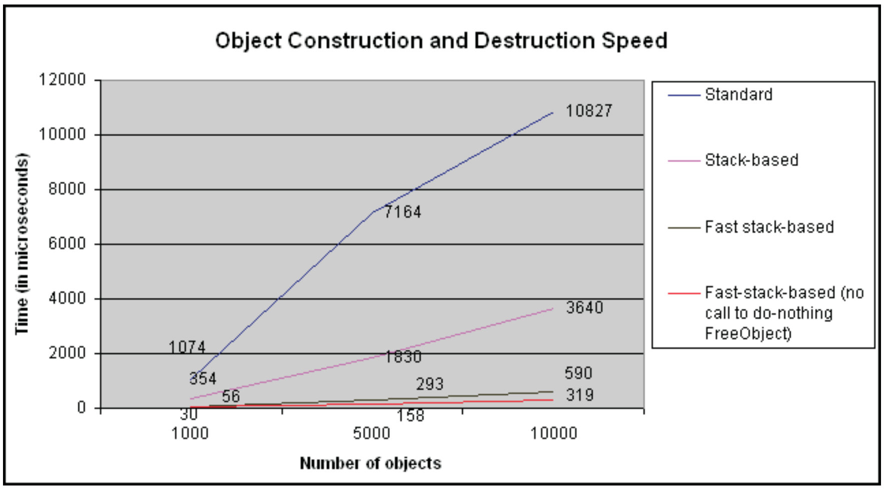
by Pedro Agulló Soliveres

The construction and destruction of small objects in Delphi can be relatively slow, especially when compared with languages such as C++. Overhead in object construction has several causes. First of all, the memory manager is called to get memory for the object, which results in `GetMem` being called. Furthermore, the VMT pointers have to be initialized: if a class does not implement interfaces, then only one VMT will be present, but for classes implementing interfaces several VMT pointers will have to be initialized. Finally, object fields will have to be initialized to zero.

Overhead in object destruction has several causes. Special fields, such as strings, interfaces, variants and dynamic arrays, will have to be cleaned up to avoid losing the memory used by them. And the memory manager will have to be called to free the object memory, resulting in `FreeMem` being called.

With all these issues, construction and destruction are far from trivial operations, despite what it might seem, due to Delphi transparently performing some *magic*.

In order to improve construction and destruction speed, we have found that the best strategy is to improve memory allocation and deallocation speed. In fact, in this article we are going to study how to avoid the whole issue by allocating objects on the stack, something that is not directly supported in Delphi. This will allow us to improve construction and destruction speed by an order of magnitude. To provide an idea of the level of optimization that we will achieve, Figure 1 shows some measurements for standard heap-based creation and destruction of small `TInteger` objects, as well as times for stack-based creation and destruction. These timings are for a 700MHz Athlon: tests for a Cyrix 200 CPU (that can be considered a very fast 486 CPU) show as much as twice these improvements.



➤ *Figure 1: Speed comparison for construction and destruction strategies.*

➤ *Listing 1*

```
// Construction pseudocode
obj := TMyClass.NewInstance
// NewInstance executes the
// following lines of code
// obj :=
//   Allocate memory(InstanceSize)
// TMyClass.InitInstance(obj);
try
  Execute code in our constructor
  obj.AfterConstruction
except
  Execute code in our destructor
end;
```

How Object Creation And Destruction Is Accomplished

In order to provide support for object allocation on the stack, we will need to understand how object creation and destruction are performed.

The first step in object creation is memory allocation, which is performed by the virtual `NewInstance` method. This calls the `InstanceSize` class method to find out how much memory to allocate, and then `GetMem` (note, however, that you can re-define `NewInstance` to provide an alternative way of allocating memory). Finally, `NewInstance` calls the non-virtual `InitInstance` class method, which sets up the object VMTs and sets the remaining memory to zero. As the last step, the code we wrote inside our constructor will be executed, and then the `AfterConstruction` virtual method will be called.

There is another thing to take into account: if an exception is raised inside the constructor, then the destructor will be automatically called to perform cleanup, and the exception will be re-raised once this has finished. Note, however, that if `NewInstance` fails to allocate memory, an `EOutOfMemory` exception will be raised, but the destructor will not be called, because nothing has been constructed yet. Furthermore, if `BeforeConstruction` fails, then `AfterConstruction` will never be called. Even when `BeforeConstruction` has successfully finished for a base class, its corresponding base class `AfterConstruction` method will not be called. Listing 1 shows the pseudocode for object construction.

When it comes to destruction, the first step is to call the `BeforeDestruction` virtual method, and then to execute the code we wrote inside `Destroy`. `CleanupInstance` is then called to perform

```
// Destruction pseudocode
obj.BeforeDestruction
Execute code in our destructor
obj.CleanupInstance
obj.FreeInstance
```

➤ *Listing 2*

cleanup of member strings, interfaces and other special types, for which Delphi manages memory under the covers. As the last step, the `FreeInstance` virtual method will be called to free the memory allocated by `NewInstance`. By default, it just calls `FreeMem`. You'll need to override `FreeInstance` if you have taken over memory allocation by overriding `NewInstance`. Listing 2 shows the pseudocode for object destruction.

Of course, the whole process is a bit more complicated, but this is a good enough description to allow us to find a way to allocate our objects in the stack. The source code in TraceableClass.pas on this month's disk performs a detailed trace of the construction and destruction process, including interaction with base class construction and destruction in the presence of exceptions.

An Auxiliary Type For Stack-Based Construction

The first thing to do to perform construction on the stack is to get additional space on it for our object. To do this, we will define an auxiliary type that has exactly the same size as our objects:

```
SizeOf(AuxType) =
  TheObjectClass .InstanceSize
```

placing a variable of that type on the stack. We will create our object on top of the space that variable occupies, as we will see later.

In our example, we will define a `TInteger` class, which has an `FValue` field, of type `Integer` (see Listing 3). The corresponding auxiliary type will be `TIntegerRec`, defined:

```
// Auxiliary type
TVMT = Pointer;
TIntegerRec = record
  _VMT : TClass;
  _ FValue : Integer;
end;
```

By providing exactly the same fields as for the object, plus an additional field at the beginning, corresponding to the object VMT, we will ensure that the defined record has exactly the same size as a `TInteger` instance (OK, that's not exact, but we will study this issue later). Anyway, and just in case we miss something, we will add an additional check in the initialization section of the unit where `TInteger` is defined:

```
Assert(SizeOf(TIntegerRec)
  = TInteger.InstanceSize);
```

This will provide a safety net just in case we add a field to `TInteger` and forget to add it to `TIntegerRec`. Additionally, when an object implements several interfaces, correctly defining the `TxxxRec` auxiliary type will be a bit more difficult, and then this check will prove invaluable. Furthermore, field layout may vary due to alignment issues, something that will be caught by this check: we will deal with and solve these issues later.

A Simple TInteger Class

Let's define a simple `TInteger` class: simply a wrapper around an `Integer`, with a constructor that receives a value, a destructor that does nothing (but is needed to illustrate how to write a destructor), and a property that allows setting and getting the `Integer` value. The declaration of this class is shown in Listing 3.

In order to make the process easier, we provide two auxiliary methods, `DoCreate`, and `DoDestroy`. If you remember the previous explanation, the code in `Create` is called as the last step in construction, while the code in `Destroy` is called as the first step in destruction: by writing their code in `DoCreate` and `DoDestroy`, we are providing an easy way of calling it without the code magically generated by Delphi being executed at the same time. The code for `TInteger` is shown in Listing 4.

Supporting On-Stack Creation

The next step is to provide support for construction and destruction of `TInteger` on the stack. To this end, we are going to define a `CreateObject` function that creates an object on top of a `TIntegerRec` variable on the stack, and a `FreeObject` function that performs the equivalent of destroying the object. Furthermore, since we are going to place the `TInteger` on top of a `TIntegerRec` variable, we will provide a `GetObject` that returns

```
TInteger = class
private
  FValue : Integer;
public
  // Just calls DoCreate
  constructor Create( value : Integer );
  // Just calls DoDestroy
  destructor  Destroy; override;
  property    Value : Integer read FValue write FValue;
protected
  // Performs all operations for Create, except memory allocation
  procedure   DoCreate( value : Integer ); virtual;
  // Performs all operations for Destroy, except memory deallocation
  procedure   DoDestroy; virtual;
end;
```

➤ *Above: Listing 3*

➤ *Below: Listing 4*

```
constructor TInteger.Create(value: Integer);
begin
  DoCreate( value );
end;
procedure TInteger.DoCreate(value: Integer);
begin
  FValue := value;
end;
destructor TInteger.Destroy;
begin
  DoDestroy;
end;
procedure TInteger.DoDestroy;
begin
  // Ok, we do nothing, but this is needed
  // for illustrative purposes
end;
```

the object given a `TIntegerRec`. The interface for these functions is shown in Listing 5.

It is of paramount importance that `Free` or `Destroy` are never called for objects created in the stack by `CreateObject`: this would end up in the memory manager holding a block of memory in the stack as a free block, due to `FreeMem` being called by `Destroy`.

Note that all methods are overloaded, so that we can write other functions with the same names for creating and destroying instances of other classes in the stack.

In order for `CreateObject` to be completely equivalent to a constructor, we will have to provide a way of allocating memory for the object, then initializing that memory to zero, as well as initializing the object VMTs, and then call the code inside the constructor and `AfterConstruction`. Furthermore, we will have to call the destructor if some exception is raised inside the constructor, so that the standard Delphi semantics are preserved. The code is shown in Listing 6.

The memory itself is provided by placing a `TIntegerRec` variable in the stack, which we receive in the `rec` parameter. First of all, we initialize the object's memory, by calling `InitInstance` passing the address of the *allocated* memory to it, that is, `@rec`. Once this is done, we get the `TInteger` created on top of `rec` by calling `GetObject(rec)`, and then call the code inside the constructor, which we have wisely placed in `DoCreate`.

```
// Equivalent to TInteger.Create, but uses the memory provided by rec,
// so that a TIntegerRec in the stack can be used
function  CreateObject( var rec : TIntegerRec; value : Integer ):
  TInteger; overload;
// Returns the TInteger allocated in rec: note that
// CreateObject(rec, value) must have been called previously.
function  GetObject( const rec : TIntegerRec ): TInteger; overload;
// Subsitutes GetObject(rec).Free. In fact, this can't
// be called because it would call FreeMem(@rec), which is not
// a good idea!
procedure FreeObject( var rec : TIntegerRec ); overload;
```

➤ *Above: Listing 5*

➤ *Below: Listing 6*

```
function CreateObject( var rec : TIntegerRec; value : Integer ): TInteger;
begin
  TInteger.InitInstance(@rec);
  Result := GetObject(rec);
  try
    Result.DoCreate( value );
    Result.AfterConstruction;
  except
    Result.DoDestroy;
    Result.CleanupInstance;
    raise;
  end;
end;
```

```
procedure FreeObject(
  var rec : TIntegerRec);
var
  obj : TInteger;
begin
  obj := GetObject(rec);
  obj.BeforeDestruction;
  obj.DoDestroy;
  obj.CleanupInstance;
end;
```

➤ *Listing 7*

```
procedure TestStackCreation;
var
  r : TIntegerRec;
  int  : TInteger;
begin
  int := CreateObject(r, 10);
  try
    ShowMessage(
      '"int" has a value of ' +
      IntToStr(int.Value));
  finally
    FreeObject(r);
  end;
end;
```

➤ *Listing 8*

If something goes wrong, the equivalent of the destructor must be called, and then the exception should be re-raised. Of course, we must not call `Free` or `Destroy`. Furthermore, `FreeObject` cannot be called, because it calls `BeforeDestruction`, which is never called for heap-based objects when the destructor is called due to an exception being raised during construction. Hence, we will have to call `DoDestroy` and `CleanupInstance` manually.

`FreeObject` is implemented so that `BeforeDestruction` is called first. Next, we execute the code in the destructor, by calling `DoDestroy`. Following that, the `CleanupInstance` routine is called. The order of these calls is important, because the code in `DoDestroy` may use special fields whose memory `CleanupInstance` may have to free. The code is shown in Listing 7.

Please note that `FreeObject` must never be called for `TIntegerRecs` for which `CreateObject` has not been called.

The last function to implement is `GetObject`, which has a trivial implementation: it just returns the address of the `TIntegerRec` it receives, as follows:

```
function  GetObject(const rec:
  TIntegerRec): TInteger;
begin
  Result := TInteger(@rec);
end;
```

Note that the code we have written will be almost the same for all classes: only the call to `DoCreate` in the `CreateObject` will have to be modified for different classes, and just to pass different parameters, thanks to our strategy of encapsulating all the code for construction and destruction in the `DoCreate` and `DoDestroy` virtual methods.

A Code Sample

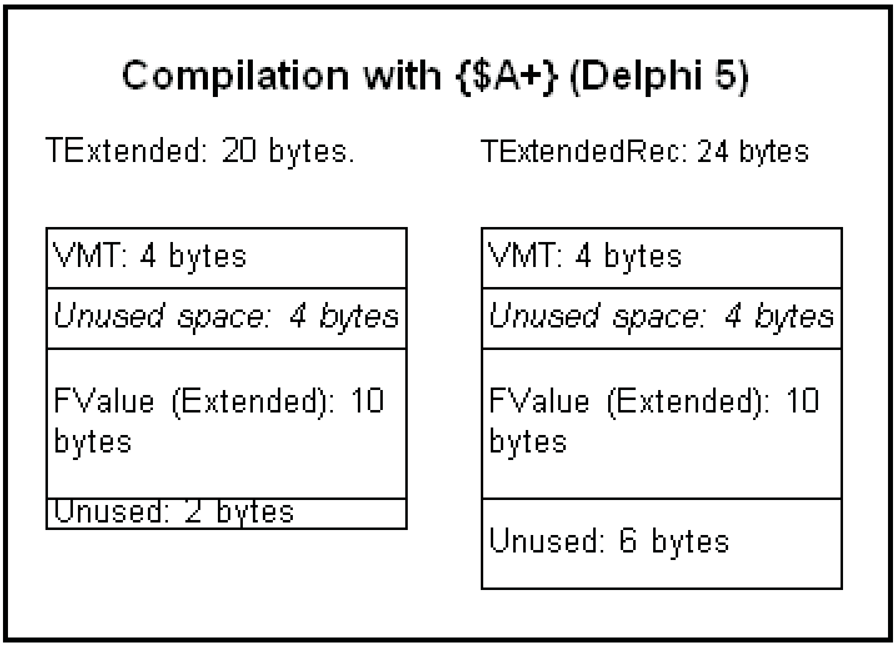
Now that we have all the pieces in place, let's write a small code sample that creates a `TInteger` on the stack and shows a message with its value. This code can be found in Listing 8. As you can see, it is pretty easy to use these stack-based objects: the only difference is that you use `CreateObject` to create them, and call `FreeObject` passing the auxiliary variable on top of which the object has been allocated, instead of using `Create` and `Free`.

Fastest Possible TInteger Construction And Destruction

It is possible to get much faster construction and destruction of a `TInteger`. To illustrate this, we will use a `TFastIntegerRec` auxiliary type, which just happens to be an exact copy of `TintegerRec`: this way, the demo program will be able to show both ways of instantiating stack-based `TInteger` objects. Listing 9 shows the code for the creation and destruction of `TInteger` objects on top of `TFastIntegerRec` types on the stack.

With respect to the new `CreateObject` version, note that we are relying on the fact that we know `TInteger` intimately: therefore, we know that `AfterConstruction` and `BeforeDestruction` are not overridden, meaning that they are do-nothing operations that we do not need to call. Furthermore, we know that there are no special fields that need magic intervention for cleanup (that is: strings, dynamic arrays or interfaces) and because of this we do not need to call `CleanupInstance` in the destructor. Also, since we know that `TInteger` has only one VMT, at the beginning of the object, we can initialize it by hand, by writing `rec._VMT := TInteger`. We do not need to set the fields' memory to zero, because we are directly initializing the only field in the object, `FValue`, and that would be redundant. By doing all of this manually, we avoid having to call `InitInstance`.

Figure 2: Layout for a TExtended and the auxiliary TExtendedRec type when compiled with the $A+ directive under Delphi 5.



```
function CreateObject(var rec : TFastIntegerRec; value : Integer): TInteger;
begin
  rec._VMT := TInteger;
  Result := TInteger(@rec);
  Result.FValue := value;
end;
procedure FreeObject(var rec : TFastIntegerRec); overload;
begin
end;
```

Listing 9

Since the destructor does nothing, we don't even need to call `DoDestroy`, and `FreeObject` ends up doing nothing. It is provided only so that the end-user can write the same kind of code for different stack-based classes, calling `CreateObject` and then `FreeObject`, the same way all heap-based object construction and destruction code is written following the same pattern, by calling `Create` and later `Free`. Of course, not calling `FreeObject` is optimal in this case.

With these optimizations in place, we have made stack-based construction more than thirty times faster than heap-based construction in a Pentium-class machine. Destruction is now more than sixteen times faster (or, if we rely on the class user knowing that there is no need to call any destruction function for `TInteger`, we will get it for free).

To better appreciate the gains we have made, we have tested how much time it takes to simply assign an integer (which is really `Integer` *construction* time): it is less than five times faster than fast `TInteger` stack-based construction. This makes `TInteger` stack-based construction a really fast operation, indeed.

```
type
  TExtended = class
    FValue : Extended;
  end;
  TExtendedRec = record
    FVMT : TClass;
    FValue : Extended;
  end;
```

Listing 10

However, be aware that writing a fast version of the stack-based construction and destruction procedures requires that you fully understand the whole construction and destruction process, and it can break down if somebody adds an `AfterConstruction` procedure, adds a *special* field (such as a string, etc), and so on.

By contrast, our first attempt at stack-based construction and destruction is almost foolproof, and won't break if changes are performed to our class. It is up to you to decide which alternative will be more convenient.

Issues With Auxiliary Types Layout

There are several issues that can affect memory layout in an object or record, potentially making the auxiliary type definition incompatible with the class instances it will hold. For example, Figure 2 shows the memory layout for a `TExtended` class and its auxiliary type for stack creation, `TExtendedRec`, when compiled under Delphi 5 with the `$A+` directive (equivalent to checking the project compiler option *Aligned record fields*).

The code for both types is shown in Listing 10. Note how the `TExtendedRec` type is larger than a `TExtended` instance (24 bytes against 20 bytes). However, when compiled with the `$A-` directive, the `TExtendedRec` type occupies 14 bytes, against the 16 bytes a `TExtended` instance uses, as seen in Figure 3.

Note that using a `TExtendedRec` variable compiled with the `$A-` directive would produce disaster when used to hold a `TExtended` compiled with the `$A+` directive, because it does not provide enough space to hold it. Of course, this won't happen, but we need a foolproof way of guaranteeing that a `TExtendedRec` variable will always be capable of holding a complete `TExtended` instance.

Differences in data type sizes and the layout of data are due to several factors. For example, the `InstanceSize` of all classes seems to always be a multiple of four, at least in Delphi 5, probably due to the granularity of the memory manager, which returns blocks of sizes that are a multiple of four. On the other hand, the memory layout for code compiled with `$A+` is modified to get faster access to data, for example, so that `Integer` data is always placed at addresses that are a multiple of 4, something the CPU likes a lot. That is the cause of types having bigger sizes than we might expect: the unused space is added by the compiler so that data is placed at boundaries where access will be much faster. For further information on this, you can read the Delphi online help, searching on *data alignment*, and especially reading the *Record types* entry.

Now, how do we guarantee that our auxiliary record type is of the appropriate size? The best workaround is just not to forget placing the following safeguard at the beginning of the initialization section in the appropriate unit:

```
Assert(SizeOf(TxxxRec) =
  Txxx.InstanceSize);
```

Then, place a breakpoint in that line and run the program inside the debugger, looking at what `Txxx.InstanceSize` returns, say `SIZE`. Once we have the instance size, we'll modify our source code so that `TxxxRec` is defined as follows:

```
TxxxRec = packed record
  FVMT : TClass;
  array [0..SIZE-1-SizeOf(
    TClass)] of Byte;
end;
```

In fact, this is probably the best approach to write the auxiliary type, because it does not allow easy back door access to the internals of our objects through the `TxxxRec` variable, a hole our `TIntegerRec` implementation left open. Note too that we can still initialize the internal `TxxxRec` VMT by just writing `rec.FVMT := Txxx`, because it is the first field in our record, and therefore not subject to *movement* by the compiler: it is guaranteed that it will always be placed at the beginning, the same way an object VMT is always placed at the beginning of the instance.

Classes implementing interfaces will have more than one VMT, and therefore manual assignment of the VMTs is much more difficult, because we have to know where each VMT is placed inside the objects: this can be done both at compile and run time, but we feel that it is not worth the effort. Just calling `InitInstance` so that the compiler copes with this issue is a much safer approach.

Other Issues

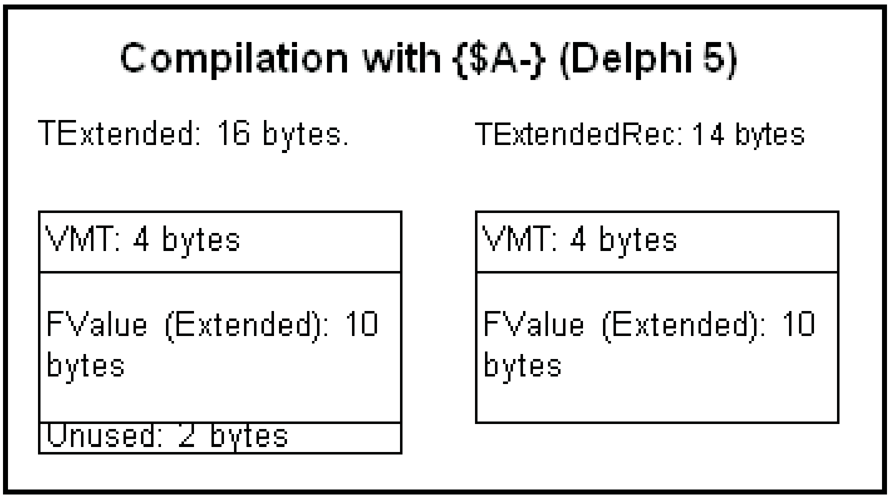
There are several things that we have omitted in the previous discussion. First of all, we have made no attempt to override `NewInstance` so that we can devise a way of allocating objects in the stack. This may be feasible, although in an indirect way, by providing the address of the stack variable in a global variable, and then returning it in `NewInstance`. However, this is not very clean, nor is it thread safe. We do not consider rewriting `NewInstance` in this way to be a good idea.

A better alternative would have been to define a function that receives a variable of the auxiliary record type, `TxxxRec`, returns an address as a `Txxx` object, and then lets the end-user call `Create`, by using the `obj.Create` syntax, as in Listing 11.

This looks like a very good idea, but it has a very important drawback: if `Create` fails, the destructor (or its equivalent) will not be automatically called, breaking the standard Delphi construction semantics. Of course, the end-user can take the responsibility for performing destruction by providing a `try...except` block, but he or she may forget to do that. It is better to place this responsibility where it belongs (on the shoulders of `TInteger`'s implementor), and write a function that does what it has to do. Of course, this is the `CreateObject` function we wrote to begin with.

Another thing to take into account is that both `DoCreate` and `DoDestroy` have been implemented as virtual. By doing this, we can override them in derived classes

➤ *Figure 3: Layout for a TExtended and the auxiliary TExtendedRec type when compiled with the $A- directive under Delphi 5.*



```
function CreateObject( var rec : TIntegerRec ):
  TInteger;
begin
  Result := TInteger(@rec);
end;
...
var
  r : TIntegerRec;
  i : TInteger;
begin
  i := CreateObject(r);
  // This looks good, but if Create fails then the destructor
  // is not automatically called
  i.Create(33);
  // ...
end;
```

➤ *Listing 11*

(typically calling the base class implementation), making support of stack-based allocation for derived classes very easy.

We have offered two ways of providing support for stack-based objects: the first was easier, and more robust, in the sense that modifications to classes supporting stack-based instantiation would not require modifications to our code. Our second approach provided much faster creation and destruction, at the price of complexity and having to rewrite code each time a class is modified. We feel that the second approach is the one to follow for small classes for which modifications are very unlikely, such as object-based wrappers for primitive types, node types, and the like, while the first approach will be more appropriate for large or unstable classes, especially if inheritance is involved.

When it comes to the applicability of the *stack allocation* idea, we would like to note that we are not limited to providing support for stack-based allocation. In fact, what we have provided here is a way of supporting *placement creation* of objects, that is, creation of objects at the address we desire: for example, we may implement a linked list with a node pool for faster allocation, with nodes that provide space both for the pointers to other nodes *and* an object instance. With the mechanisms outlined here we can create instances on top of the node, just after the pointer fields. In fact, we have implemented such a kind of list, which outperformed other lists by a significant factor (more than three times faster than the best competing list), with much less memory wastage.

Acknowledgements

I would like to thank my colleagues at IPS, Francesc Folguera, Oscar Carrera, Miquel Angel Carrera and Jordi Costa, for their suggestions, as well as for providing a superb and enjoyable working environment during the past years.

Pedro Agulló Soliveres is a freelance consultant, programmer and technical journalist specialising in Delphi, C++ and Java.