Soletta

STANDARD DELPHI LIBRARY

TUTORIAL AND REFERENCE GUIDE

STANDARD DELPHI LIBRARY

TUTORIAL AND REFERENCE GUIDE

Soletta
2103 Patty Lane
Vienna, VA 22182

http://www.soletta.com

Please email support@soletta.com with any technical questions about this product.
Email sales@soletta.com with any sales questions.

TABLE OF CONTENTS

Introduction ____ *1*

Product Versions ____ **1**

Ordering SDL and/or SuperStream ____ 2

How to use this documentation ____ *4*

Quick Start ____ *5*

Installation ____ *6*

Archive Installation ____ **6**

Soletta Store Installation ____ **6**

Basic Concepts ____ *7*

Accessing the SDL and SuperStream Libraries ____ **7**

Item ____ **7**

Container ____ **7**

Iterator ____ **7**

Comparator ____ **7**

Closure ____ **8**

Morphing Closure ____ **8**

Garbage Collection ____ **8**

AtEnd ____ **8**

Range ____ **8**

Pair ____ **8**

Sequence ____ **9**

Vector ____ **9**

Map ____ **9**

Set ____ **9**

Hashing ____ **9**

Algorithm ____ **10**

Quick Example ____ *11*

A Note About Namespaces ____ *12*

Error Handling ____ *13*

Ten Easy SDL Lessons ____ *14*

Lesson 1 - Keeping Lists of Objects ____ **14**

Lesson 2 – Keeping Lists of Strings and other Atomic Types ____ **15**

Lesson 3 – Iterating with Iterators ____ **16**

Lesson 4 – Using SDL instead of Delphi's Data Structures ____ **18**

Lesson 5 – Using Maps (Key-Value Pairs) 19
Lesson 6 – Using Sets 21
Lesson 7 – Using the Sort Algorithms 22
Lesson 8 – Changing Data Structures 23
Lesson 9 – Transforming Objects 25
Lesson 10 – Filtering Objects 25
Containers *27*
All About DObjects 27
Example Code 28
Container hierarchy 28
DIterator 29
Forward Iterators 31
Bidirectional Iterators 32
Random Iterators 32
Iterator Adapters 32
DContainer 32
Comparators 33
Constructing Containers 34
Number of Items 34
Adding Items 34
Removing Items 35
Retrieving Items 35
DSequence 36
Adding Items 36
Retrieving Items 36
Removing Items 36
DVector 36
DAssociative 37
Sets and Maps 37
Adding Elements 37
Finding Elements 38
Removing Elements 38
Container Adapters 38
Creating Your Own Containers 38
Frequently Asked Questions 38
How do I get the number of items in a container? 38
How do I add items to a container? 38
How do I iterate over a container? 39
How do I retrieve the keys from a map container? 39
How do I sort a sequence? 39
Why does SDL use functions instead of class members for its algorithms, and for iterator operations? 39
How do I find items in a map? 40
Algorithms *41*
A Note About Ranges 41

Naming Conventions 41

Applying 42
forEach 42
Inject 42

Comparing 42
Equal 42
LexicographicalCompare 43
Median 43
Mismatch 43

Copying 44
Copy 44

Counting 44
Count 44

Filling 44
Fill 44
Generate 45

Filtering 45
Unique 45
Filter 45

Finding 45
AdjacentFind 45
BinarySearch 46
Detect 46
Every 46
Find 46
Some 47

Freeing and Deleting 47
ObjFree 47
ObjDispose 47
ObjFreeKeys 47

Hashing 47
OrderedHash 47
UnorderedHash 47

Removing 48
Remove 48
removeCopy 48
removeIf 48

Replacing 49
Replace 49
ReplaceCopy 49
ReplaceIf 49

Reversing 49
Reverse 49
ReverseCopy 50

Rotating 50
Rotate 50
RotateCopy 50

Set Operations 50

Includes 50
SetDifference 50
SetIntersection 51
SetSymmetricDifference 51
SetUnion 51

Shuffling 51
RandomShuffle 51

Sorting 52
Sort 52
StableSort 52

Swapping 52
IterSwap 52
SwapRanges 52

Transforming 52
Collect 52
TransformBinary 53
TransformUnary 53

Utility Functions 54

Atomic Converters 54

Iterator Helpers 54

Hashing 54

DObject Helpers 55

Morphing Closures 56

Printing 57

Debugging Support 59

Persistence with SuperStream 60

Basic Concepts 60
Stream 60
Object 60
Atomic Types 60
Transfer Function 61
Object Versioning 61
Buffered Stream 61

Nine Easy SuperStream Lessons 61
Lesson 1 – Saving and Loading One Object 62
Lesson 2 – Storing Different Objects 64
Lesson 3 – Writing Embedded Objects 65
Lesson 4 – Inheritance and SuperStream 66
Lesson 5 – Storing SDL Containers 68
Lesson 6 – Storing Special Types (TDateTime, Single, Double) 68
Lesson 7 – Storing Raw Data 71
Lesson 8 – Storing Complex Object Graphs 72
Lesson 9 – Reading and Writing Different Versions of Objects 73

SuperStream Classes 75
TStreamAdapter 75
TObjStream 75

TBufferedInputStream 76
TBufferedOutputStream 76
TObjList 76

Epilogue ***77***

Introduction

The Standard Delphi Library (SDL) by Soletta is a powerful library of reusable container classes, generic algorithms, and an easy to use persistence mechanism. SDL is designed for intermediate to advanced Delphi programmers who have a need for sophisticated data structures or who wish to take advantage of SDL's large library of generic algorithms. SDL is also highly appropriate for programmers who are experienced with the C++ STL (Standard Template Library), or ObjectSpace's JGL (Java Generic Library). SDL is an adaptation of Stepanov and Lee's concepts to the Delphi environment.

SDL offers a number of features not found in any other Delphi class library:

- **Powerful underlying methodology**. SDL is the first data structure and algorithm library for Delphi to be based on generic programming with reusable algorithms and data structures. It is based on a mature and sophisticated model (STL).
- **Natural and easy to use storage of atomic data types**. This means that SDL containers can be used to hold any Delphi data type (such as Integers, Strings, Extended values) with no special syntax. SDL is the first container library to take advantage of Delphi's *array of const* feature, which lifts a significant burden from the programmer.
- **Generic algorithms**. Like STL and JGL, SDL comes with over 60 generic algorithms. The set of algorithms was originally chosen by Stepanov; SDL provides implementations of many of the algorithms found in STL. Thus, an STL programmer will be immediately familiar. SDL's container classes follow the interface/iterator model; the consequence is that most generic algorithms will work on any container class.
- **Integrated persistence**. SDL's companion library, SuperStream (included with SDL Source Edition), provides a capable, easy to use method for storing and retrieving objects in Delphi. SDL comes complete with the integration code necessary to use SuperStream and SDL together. SuperStream and SDL are based on many of the same techniques, so programmers who become familiar with one will be immediately familiar with the other.
- **Complete set of data structures**. SDL includes arrays, double-linked lists, maps, and sets. The mapping structures are available both in red-black tree and hashing form. There are at least ten different data structures available, with more being developed.
- **Atomic, associative data structures**. SDL is the first data structure library for Delphi to provide natural, atomic storage of associations. For example, adding to a map is as simple as map.putPair([10, 'hello']). This places the value 'hello' at key 10 in the map. Note that the values are specified without any object wrappers. We can also just as easily add objects to the map: map.putPair(['Ross Judson', objTest]).

Product Versions

SDL is available in two versions: The binary release and the source release. The garbage collection feature is available only in the source release, as it requires recompilation of the

SDL unit to enable. Both versions are available directly from Soletta – please email sales@soletta.com for information on purchasing either product. If you have the binary release and wish to upgrade to the source release, please contact Soletta, and we'll guide you through the process.

You may have acquired the trial version of SDL on the internet. If so, we're glad that you're taking the time to look at SDL, and see if it can help you. You'll find the Lessons in the sections ahead to be particularly insightful, and we recommend that you read them closely. SDL isn't really like other container class libraries for Delphi, and to appreciate the power it gives you, you'll need an open mind!

If you are coming from an STL or JGL background, reading the SDL documentation will be easy for you. SDL uses the same terms, the same words, and the same ideas. You'll find it to be an effective adaptation of the STL methodology to Delphi. You'll also find that it makes migrating C++ programmers (who have experience with STL) to Delphi easier.

SDL pushes the envelope with Delphi, right to the limit. It's exciting when a language can be manipulated to effectively accomplish tasks it wasn't designed to do directly. This is the hallmark of the mature language and environment, and Delphi's time is now.

Ordering SDL and/or SuperStream

You can purchase this package several ways: By credit card over the phone, credit card over the web, credit card through email, or check/credit card through postal mail. To order **SDL** over the internet, visit the following web page:

http://www.shareit.com/programs/101667.htm

To order **SuperStream**, visit this page:

http://www.shareit.com/programs/101670.htm

This web ordering service is provided by ShareIt!, who handle **orders only**. Please do not telephone or email ShareIt with support questions or sales questions, unless they are payment related.

If you would like to order by telephone, please call the following number:

In the USA: **1-800-903-4152**
Everywhere else: **+49-221-240727**9

During ordering please reference program # 101667 for SDL, and program # 101670 for SuperStream.

You can also order by credit card directly from Soletta. A file with ordering information is included in the distribution (**ordering.txt**).

Send email to sales@soletta.com, including the following information:

Name
Address

City, Province/State, Postal Code/Zip
Country
Name on Credit Card
Credit Card Type (Visa/MC/Amex)
Credit Card Number
Credit Card Expiry
Email address (very important!)
Product (SDL Source/Binary, SuperStream Source/Binary)
Delphi version (3 or 4)

If you prefer postal mail, you can send the information listed above to:

Soletta
2103 Patty Lane
Vienna, VA 22182
USA

How to use this documentation

SDL is sufficiently different from other container class libraries (and just about every other class library) for Delphi that you should read the **Guide** section of this manual fairly thoroughly. Programming with containers and generic algorithms takes some thought, planning, and knowledge if you're going to get the most benefit from it. Reading the **Reference** material will give you a good feel for what's in the library; you should probably read it after you've done some programming with SDL.

The Guide has three major sections – Containers, Algorithms, and Persistence. Containers discusses the data structures in SDL, enumerates their relative advantages and disadvantages, and illuminates the iterator concept. Algorithms lists the major types of algorithms that are present in SDL, and how to apply them to containers. Persistence discusses storing and loading objects from streams.

The Containers section can be read on its own. The Algorithms section should be read only after reading the Containers section. The Persistence section stands alone, but if you want to understand the mechanism used to serialize SDL Containers, you should have a working knowledge of them first.

Soletta highly recommends that you study the Algorithms at length. Their names are generic, but the tasks that they perform occur over and over again in common programming situations. The trick is to recognize when a task can be performed by one of the generic algorithms. A side effect of using them is that your code becomes more readable to persons fluent with generic algorithms. Much like patterns, code comprehension is often based on vocabulary, and the Algorithms in this library provide an excellent set of verbs.

Included with SDL is an HTML reference to the library that was generated with another Soletta product, DelphiDoc. That HTML reference should be considered to be the most accurate source of information, as it is directly generated from the SDL source code. If there are differences between what you read here and what you find in the HTML reference, this document will defer to the HTML.

Quick Start

Here's what you need to know to get started with SDL quickly:

1) Become familiar with the basic concepts of this library.

2) Learn about the basic container classes and DObjects, which are the basic items that are stored in containers.

3) Learn about iterators – the types of iterators, and the functions that operate on them.

4) Learn basic techniques for adding items to containers, and how to iterate over containers.

5) Learn about what the algorithms can do for you.

6) Learn about SuperStream, SDL's persistence mechanism.

Each of these is in the order we suggest you learn. The best thing to do is put together a few quick programs that make use of basic SDL features so you can get a feel for how the library operates. After you've done that, come back to the reference material, which you'll then be able to read with good basic proficiency in place.

Installation

SDL and SuperStream have been delivered to you in one of several ways:

- As an archive, downloaded from the internet
- On a CD-ROM
- From the Soletta Store, for online purchasing of Soletta libraries

Archive Installation

If you've downloaded an archive, unzip the contents of the archive into a new directory. Make sure that you preserve the paths in the archive! The HTML documentation has a large number of files, and you'll want to make sure that they land in the correct directories.

Make sure the directory you created is on your library path (Tools | Environment Options, library page) – Delphi needs this to find the SDL and SuperStream unit files.

Copy the sdlhelp.hlp and sdlhelp.cnt files to the Help directory in your Delphi installation. Add *:link sdlhelp.hlp* to the last section of the **delphi.cfg** file, and *:include sdlhelp.cnt* to the last section of the **delphi.cnt** file. This will enable F1 based help for SDL.

Soletta Store Installation

The Soletta Store will automatically emit libraries into a directory you specify. You will still need to add the directory to the Delphi Library Path, and add the help file entries, as above.

Basic Concepts

To effectively use SDL, you need to learn about its parts and become familiar with SDL's vocabulary. SDL uses the same words to describe its concepts that JGL and STL do, so if you're familiar with those, you'll adapt to SDL quickly. If you're not familiar with other generic programming packages, you'll want to read this section thoroughly.

Accessing the SDL and SuperStream Libraries

To use classes and functions from the SDL library, make sure the **SDL** unit is in your *uses* statement. To use classes and functions from SuperStream, make sure **SuperStream** is in your uses clause.

If you will be streaming SDL containers with SuperStream, make sure your program includes the **SDLIO** unit. No function calls are necessary – including the unit will perform all necessary registration and initialization.

Item

In SDL, items are **atomic values** or **objects**. An atomic value is one of the basic types, like an Integer, String, Currency, or Char. An object is also an atomic type because Delphi treats all objects by reference (pointer).

Container

A container is a data structure that can hold a number of items. Different types of containers have different capabilities – for example, one type of container may support very fast deletion, but slow addition, and another might support fast random access but slow deletion. When you need a container class, choose an appropriate one based on what you need that container to do. Each container class is describe later on in this guide, and the strengths and weaknesses of each are provided.

Iterator

An iterator is analogous in many ways to a pointer. It points at a certain item in a container. Iterators can be moved forward, and can usually be moved backwards. The object under the iterator can be retrieved, and can sometimes be set. Iterators are the preferred way for algorithms and for your code to deal with SDL containers. If you use iterators to access the containers, you can *change the container without changing your code*. This is one of the primary strengths of SDL. You will see the following constructs very often when perusing SDL-based code:

```
Iterator := container.start;
Iterator := container.finish;
```

The **start** function retrieves an iterator positioned on the first item in the container. The **finish** function retrieves an **atEnd** iterator (which is positioned just after the last item in the container). See *atEnd* for more information on the special atEnd iterator.

Comparator

A comparator is a function used to compare two items. It should return less than zero if the first object is less than the second; zero if the two objects are equal, and greater than zero if the second object is greater than the first. Comparators are closures (procedures of objects)

with a special signature. See **Closure** and **Morphing Closure** for related information. Here's the signature of a DComparator:

```
DComparator = function (const obj1, obj2 : DObject) : Integer of object;
```

Closure

A closure is a *procedure of object*. Closures are at the heart of Delphi's event model – hopefully, as a Delphi programmer, you understand how they work. While effective for event handling, the closure mechanism is an elegant solution for any situation in which methods of objects must be called. All of SDL's functional types are defined as procedure of object. This allows them to be methods on objects. If SDL did not do this, you would need to create unit-level procedures for procedures you wanted to pass to SDL. SDL also supports the transformation of unit-level procedures into Closures – see Morphing Closure.

Morphing Closure

A unit-level procedure can be transformed into a closure by making use of one of the MakeXXX family of functions. This makes use of a Delphi trick that fools Delphi's method calling mechanism into believing that it is calling a closure.

Garbage Collection

Garbage collection is a more advanced, automatic method of dealing with memory allocation issues. Delphi has traditionally been programmed using manual memory management, which means that the programmer is responsible for allocating and deallocating all objects. In a garbage collecting system, the programmer only allocates objects. The system will deallocate them when it determines that it is permissible to do so. SDL is compatible with a garbage collection system called the **Boehm Collector**. The Boehm Collector is a conservative, mark-and-sweep collector.

AtEnd

SDL maintains positions in containers using iterators. There is a special iterator position known as **atEnd**. An atEnd iterator is positioned *one past the last item in the container*. It is illegal to retrieve an object from this position. It is sometimes legal to write to this position – certain containers will add the object being written to the end of the container, but not all containers will do this. Notably, the mapping containers will not. **AtEnd** is important – when algorithms don't succeed, they will often return atEnd as their result.

Range

A range is a pair of iterators, marking the beginning and ending of a set of items. For example, there is a range of items between **container.start** and **container.finish**.

Pair

A pair is two items (two DObjects, stored in a DPair). The mapping containers store pairs, each consisting of a key (stored in the .first field) and a value (stored in the .second field).

Sequence

A sequence is a container where the items in the container have a defined order. Containers descended from sequence will retrieve their items in the order that they were added. Examples of sequences include linked lists and arrays.

Vector

A vector is a container whose items are *numerically addressable*. That is, you can specify that you want the first item, or the tenth, or the fiftieth. While all sequences can return the item at a specific position, being a vector implies that this operation is **efficient**. Vectors are usually implemented as arrays, but other implementations can exist.

Map

Maps store key-value pairs. As a user of SDL, you should pay special attention to maps because it has been estimated that for other, similar container libraries, up to 90% of all container usage is of map-based structures. Maps store *associative* data. For example, you may want to keep a set of employee objects, keyed by employee ID. What does it mean to *key on employee ID*? It means that you want to associate an employee object (say, Jim Smith's), with a numeric ID (like 1001). You do this by putting a *pair* to the map structure:

```
Map.putPair([1001, jimSmithObject]);
```

When you want to retrieve the employee associated with 1001, you do the following:

```
employee := getObject(map.locate([1001])) as TEmployee;
```

Mapping containers ensure that the process of looking up a key value is very efficient. SDL has two basic kinds of maps in it: An ordered map (based on red-black trees) and an unordered map (based on hash structures).

There are two basic variants on maps: A MultiMap and a regular Map. The difference is that MultiMaps can store multiple objects on the same key. Storing a value at a key in a regular map will replace whatever value was stored there before.

Set

Sets store items and allow you to rapidly determine if a set contains a particular item or not. You may already be familiar with Delphi's set types. SDL's sets are much more general – you can have sets of numbers, strings, objects, or just about anything else. As with maps, there are MultiSets and regular Sets. A MultiSet can have multiple copies of an object in it. Sets also come in the two basic kinds – red-black based and hash based.

Hashing

Hashing is the process of converting an item (or object) into a number. SDL provides a number of hashing functions and makes use of hashing internally. Creating good hash functions is difficult – ideal hash functions try to create very random-looking numbers from whatever objects are given to them. Making use of SDL's hash functions ensures that you are getting good hash performance.

Algorithm

An algorithm is a series of steps necessary to carry out a process. Most algorithms operate on data, make decisions about what to do based on that data, and transform the data into some kind of output. SDL contains a large number of reusable algorithms. Reusable algorithms are solutions for problems that crop up again and again in common programming situations. By learning about SDL's algorithms, you can avoid writing a lot of common code, and simply substitute the appropriate reusable algorithm.

Quick Example

Many Delphi programmers may not be familiar with generic programming. Before going into great detail about containers, algorithms, and all the other SDL features, let's look at an example of SDL programming. First we'll create a narrative sentence that describes what we want to do. Then we'll present SDL-based code that does it. Once you see how compact the SDL code is and how it solved the problem, you'll want to know more about SDL! Here is our narrative:

We have two classes of students. Some students can be in both classes. We want to find every student whose grade is above 80 in both classes, making sure that we remove duplicates (because students might be in both classes). Then we want to sort the students by their names, in reverse alphabetical order. Here's the code:

```
Procedure test;
Var class1, class2 : DMap;
   GoodStudents : DArray;
    I : Integer;
    Iter1, Iter2 : DIterator;
Begin
  // fill our classes with random students and grades
  class1 := DMap.Create;
  class2 := DMap.Create;

  for I := 1 to 25 do
    begin
      class1.putPair([Random(100), RandomName]);
      class2.putPair([Random(100), RandomName]);
    end;

  goodStudents := DArray.Create;
  iter1 := class1.lower_bound([80]);
  iter2 := class2.lower_bound([80]);
  setIntersectionIn(iter1, class1.finish, iter2, class2.finish, goodStudents.finish);
  reverse(goodStudents);

  PrintContainer(goodStudents);

  FreeAll([class1, class2, GoodStudents]);

End;
```

Notice how compact the code is! The key to using SDL effectively is learning about the special algorithms it gives you and applying those algorithms in your programming.

A Note About Namespaces

SDL uses some rather common (and short) names for certain procedures and functions. An early design decision was taken to *not* prefix all SDL items with a special tag, such as *SDL_*. First, such a convention requires that they be absolutely everywhere, and would seriously impact the readability of the code. Second, Delphi's namespace (unit) rules are quite straightforward.

We chose instead to place all SDL functionality in a single unit. Any time a namespace conflict is discovered, simply prefix the desired SDL call with SDL. (that is SDL period). Here is an example:

```
Advance(iter);
SDL.Advance(iter);
```

These represent exactly the same function call.

In order to avoid naming collisions for classes, we have chosen to use the letter D (as opposed to T) to prefix all SDL class names. That is why you see DObject, DContainer, and so forth.

We hope that this is acceptable to you. If you have suggestions about how to modify this scheme, please email us. Of course, purchasing the source edition will allow you to make any changes you want.

Error Handling

SDL and SuperStream throw exceptions whenever illegal conditions are encountered. SDL's exceptions are rooted by SDLException. You should never rely on exception handling during the normal course of execution in your program. Therefore, any SDLException throw by your program should be considered a bug that must be eradicated. Use SDL's various testing methods to ensure that a method call will succeed before you execute it.

SDL also has a number of assertions throughout its implementation. If you purchased the source version, you can compile a version of SDL that has these assertions enabled, which provides additional diagnostics.

Ten Easy SDL Lessons

Because Delphi programmers may not be familiar with SDL programming techniques, we present ten examples here, showing how SDL's generic algorithms and containers can be used. For each example we'll provide a short narrative describing the problem and then display the SDL-based code that is a solution.

All of the lessons will use this simple object definition:

```
Type
  TEmployee = class
    Id : Integer;
    Name : String;
    Salary : Integer;
    Benefits : Boolean;
  End;
```

Lesson 1 - Keeping Lists of Objects

Let's say that we want to store a list of employee objects. This is very simple to do with Delphi's own TList class, and it's something that nearly every Delphi programmer has done. We're going to create a list of employee objects, then print out the salary values for each one. Let's look at the Delphi code, and then see the equivalent SDL code.

```
Procedure DelphiList;
Var list : TList;
    I : Integer;
    Emp : TEmployee;
Begin
  For I := 1 to 10 do
    List.add(TEmployee.Create);

  For I := 0 to list.count – 1 do
    Begin
      Emp := TEmployee(list[I]);
      Writeln('Salary for ', emp.name, ' is ', emp.salary);
    End;

  For I := 0 to list.count – 1 do
    TObject(list[I]).free;

  List.free;
End;
```

That's pretty simple code. The SDL code is just as simple, and we'll see later on how much flexibility using SDL gives you.

```
Procedure SDLList;
Var list : DList;
    I : Integer;
    Iter : DIterator;
Begin
  List := DList.Create;
  For I := 1 to 10 do
    List.add([TEmployee.Create]);
```

```
  Iter := list.start;
  While not atEnd(iter) do
    Begin
      Emp := getObject(iter) as TEmployee;
      Writeln('Salary for ', emp.name, ' is ', emp.salary);
      Advance(iter);
    End;
  ObjFree(list);
  List.free;
End;
```

The SDL code, while structured very slightly differently, is quite easy to read. Note the use of the ObjFree function – you'll learn a lot more about the many functions (or **algorithms**) that SDL offers later on. Here is another way of writing the same thing, in a more SDL-centric way:

```
Function GenEmployee(ptr : Pointer) : DObject;
Begin
  Result := Make([TEmployee.Create]);
End;
Procedure PrintEmployee(ptr : Pointer; const obj : DObject);
Begin
  With asObject(obj) as TEmployee do
    Writeln('Salary for ', name, ' is ', salary);
End;
Procedure WriteEmployee(ptr : Pointer; const obj : DObject);
Var list : DList;
Begin
  List := DList.Create;
  Generate(list, 10, MakeGenerator(GenEmployee));
  ForEach(list, MakeApply(PrintEmployee));
  ObjFree(list);
  List.free;
End;
```

Note how compact the WriteEmployee procedure is, and how clearly it reads. We are using three algorithms here – generate, forEach, and ObjFree. Generate calls a *generator function*, which is a function that creates DObjects. ForEach calls a function with each item in a container, and ObjFree calls TObject.Free for each item in a container. These three algorithms are just a small part of the many generic algorithms that are part of SDL. Using these algorithms creatively is the key to multiplying your productivity.

Lesson 2 – Keeping Lists of Strings and other Atomic Types

One of the best things about SDL containers is that they don't hold just pointers, or objects, like other data structures for Delphi do. They can hold just about any atomic type. And you can even mix them in the same container! Let's say that we want to store a bunch of strings, numbers, and floating point values in a container. Here's how we can do that:

```
Procedure GenMix;
Begin
  Case Random(3) of
  0 : result := Make([Random(10)]);
  1 : result := Make(['str ' + IntToStr(Random(10))]);
  2 : result := Make([Random(1000) / 1000]);
```

```
  end;
end;
procedure PrintMix(ptr : Pointer; const obj : DObject);
begin
  case obj.vtype of
    vtInteger: writeln('Integer: ', asInteger(obj));
    vtAnsiString: writeln('String: ', asString(obj));
    vtExtended: writeln('Extended: ', asExtended(obj));
  end;
end;
Procedure MixEmUp;
Var a : DArray;
Begin
  A := DArray.Create;
  Generate(a, 10, MakeGenerator(GenMix));
  ForEach(a, MakeApply(PrintMix));
  a.free;
end;
```

SDL can, in its containers, effectively handle a mixture of atomic types. Most of the time you'll just store one type in a container, but it's nice to know that the flexibility is there. SDL stores the following atomic types:

- VtInteger
- VtBoolean
- vtChar
- vtExtended
- vtString
- vtPointer
- vtPChar
- vtObject
- vtClass
- vtWideChar
- vtPWideChar
- vtAnsiString
- vtCurrency
- vtWideString

SDL takes care of the storage of all of these atomic types automatically – although it's still important for you to understand what's going on underneath, so that you can manipulate the DObject values correctly. You'll learn more about this later.

Lesson 3 – Iterating with Iterators

Iterators are one of the most powerful features in the SDL library. By making extensive use of iterators, you're insulating yourself against changes in your program's structures. Iterators work the same way across all SDL containers. What follows is an example that demonstrates this, by storing the same data in both a list structure and in an array structure, and performs the same operations on both.

```
Procedure Iteration;
```

```
Var iter : DIterator;
    Arr : DArray;
    List : DList;
    Sum : Integer;
Begin
  Arr := DArray.Create;
  List := DList.Create;
  Generate(arr, 10, MakeGenerator(GenEmployee));
  CopyTo(arr, list);

  Sum := 0;
  Iter := arr.start;
  While not atEnd(iter) do
    Begin
      Inc(sum, TEmployee(getObject(iter)).salary);
      Advance(iter);
    End;
  Writeln('Sum is ', sum);

  Sum := 0;
  Iter := list.start;
  While not atEnd(iter) do
    Begin
      Inc(sum, TEmployee(getObject(iter)).salary);
      Advance(iter);
    End;
  Writeln('Sum is ', sum);

  ObjFree(arr);
  Arr.free;
  List.free;
End;
```

Note that the code to iterate over the list and the array is identical. You should also note that we only performed ObjFree on the arr variable, and not on the list variable. This is because the two containers have the same objects in them – the copyTo routine only makes copies of the pointers to the objects (it is a "shallow" copy of a the arr container).

This example demonstrates a use of iterators. There is another way of expressing the same operation using generic algorithms:

```
Function SumSalary(ptr : Pointer; const obj1, obj2 : DObject) : DObject;
Begin
  Result := make([asInteger(obj1) + TEmployee(asObject(obj2)).Salary]);
End;
Procedure UseGeneric;
Var arr : DArray;
Begin
  Arr := DArray.Create;
  Generate(arr, 10, MakeGenerator(GenEmployee));
  Writeln('Salary sum is ', asInteger(Inject(arr, [0], SumSalary)));
  Arr.free;
End;
```

By now you may be understanding why generic algorithms are so powerful! Let's look at using iterators to limit a range of an operation to a particular part of a data structure.

```
Procedure LimitGeneric;
Var arr : DArray;
   Starting, ending : DIterator;
Begin
  Arr := DArray.Create;
  Generate(arr, 10, MakeGenerator(GenEmployee));
  Starting := arr.start;
  AdvanceBy(starting, 2);
  Ending := arr.finish;
  RetreatBy(ending, 2);
  Writeln('Salary for some employees is ',
    AsInteger(
      InjectIn( starting, ending,
                [0], SumSalary)));
end;
```

Lesson 4 – Using SDL instead of Delphi's Data Structures

Delphi provides two basic data structures – TList and TStringList. Let's look at how to use SDL instead of these, and what the SDL equivalents offer in additional functionality.

Here's a simple example of TStringList code:

```
Procedure TStringThing;
Var sl : TStringList;
    I : Integer;
Begin
  Sl := TStringList.Create;
  For I := 1 to 20 do
    Sl.add(RandomString);
  Sl.sort;
  For I := 0 to sl.count - 1 do
    Writeln(sl[I]);
  Sl.free;
End;
```

Here's an equivalent SDL version:

```
Procedure SDLStringThing;
Var arr : DArray;
    Iter : DIterator;
Begin
  Arr := DArray.Create;
  For I := 1 to 20 do
    Arr.add([RandomString]);
  Sort(arr);
  Iter := arr.start;
  While iterateOver(iter) do
    writeln(getString(iter));
  arr.free;
End;
```

This version makes use of a neat SDL helper function – iterateOver. Iterate over returns true while its source iterator is not atEnd, and automatically advances it. You need to call iterateOver with a fresh iterator (not one that's already been used with iterateOver) for implementation reasons, but it's a very handy function.

TStringList also offers the very convenient IndexOf and Find functions. Here's some equivalent SDL code:

```
Procedure SDLFinding;
Var arr : DArray;
    Loc : DIterator;
Begin
  Arr := DArray.Create;
  Generate(arr, 20, RandomString);
  Loc := find(['toaster']);        // linear search
  If not atEnd(loc) then
    Writeln('found it: ', getString(loc));
  Sort(arr);
  Loc := BinarySearch(arr, ['toaster']);  // log N search
  If not atEnd(loc) then
    Writeln('found it: ', getString(loc));
  Arr.free;
End;
```

This code performs a search on an array using two different algorithms – find and binarySearch. Find is a linear search through any container, looking for a value. BinarySearch relies on the container being sorted, and performs a Log N efficiency search.

Other data structures, such as Maps, offer powerful searching and location functionality as well. Find will always work on any container, although it may not be optimally efficient.

Lesson 5 – Using Maps (Key-Value Pairs)

Studies have indicated that, when available, maps make up some 90% of all container class usage. There's a reason for that – they're amazing useful, so much so that when you stop and analyze a given storage requirement in an application, it's almost always easily phrased in terms of maps.

Until SDL, there hasn't been an effective way of storing maps. The only possibility was to use TStringList, keep it sorted, and put objects in the Objects property. There are serious problems with this approach, though – TStringList is based on arrays, and does not scale effectively. In addition, data that has a bad storage pattern (already sorted) may generate extremely inefficient results when used with TStringList. That's not to say that TStringList is inefficient – it isn't, but it is limited by the underlying data store.

SDL's mapping structures are efficient and scale well. The ordered maps, in particular, are very well suited to just about any pattern of data access: They are red-black trees, and rebalance themselves automatically to match the data stored. They maintain their good characteristics at all times, which means a guaranteed Log N time for just about any operation.

Let's create a map of employee id to employee object. This type of operation is very common – we want to be able to quickly look up any employee given his/her employee number. We want to be able to identify if we're using a particular employee number. We may also want to be able to iterate over the employees.

```
Procedure MapEmployees;
Var map : DMap;
    Iter : DIterator;
```

```
    I : Integer;
    Emp : TEmployee;
Begin
  Map := DMap.Create;
  For I := 1 to 20 do
    Begin
      Emp := TEmployee.Create;
      Map.putPair([emp.id, emp.name]);
    End;

  // locate employee with id 1001
  Iter := map.locate([1001]);
  If atEnd(iter) then
    Writeln('Employee doesn''t exist')
   Else
     Writeln('Employee is ', TEmployee(getObject(iter)).name);

  // remove employee 2004 - this will remove both the key and value.
  map.remove([2004]);

  Iter := map.start;
  While iterateOver(iter) do
    Writeln(TEmployee(getObject(iter)).name);

  // iterate over the employee ids
  iter := map.start;
  setToKey(iter);
  while iterateOver(iter) do
    writeln('Employee ID is ', TEmployee(getObject(iter)).id);

  ObjFree(map);
  Map.free;
End;
```

There are a couple of interesting things to note about this example – first, note the usage of the *locate* function to find out whether a given key is in the map or not. Second, note that the *remove* function can be called to take a key-value pair out of the map. Third, when we wanted to iterate over the *key part* of each pair, we called *setToKey* on the iterator. Calling setToKey tells SDL that when we use a getXXX function on the iterator, we want it to return the key part of a key-value pair. To set the iterator back to returning values, call *setToValue*.

Let's look at another way of mapping our employees. This time we'll do it by name. We're going to create a map of names to employee objects.

```
Procedure MapEmployees;
Var map : DMap;
    Iter : DIterator;
    I : Integer;
    Emp : TEmployee;
Begin
  Map := DMap.Create;
  For I := 1 to 20 do
    Begin
      Emp := TEmployee.Create;
      Map.putPair([emp.name, emp]);
```

```
    End;

  Iter := map.locate(['ted jones'];
  If not atEnd(iter) then
   Begin
       Emp := getObject(iter) as TEmployee;
     Writeln('Found Ted Jones, whose id is ', emp.id;
    End;

  Iter := map.start;
  While iterateOver(iter) do
    Begin
      SetToValue(iter);
      Emp := getObject(iter) as TEmployee;
      SetToKey(iter);
      Writeln('Found at key ', getString(iter), ' employee id ', emp.id);
    End;

  ObjFree(map);
 Map.free;
End;
```

Lesson 6 – Using Sets

Sets are another very common data structure. Delphi and Object Pascal provide an elegant, albeit limited, set feature in the language. Programmers coming from other languages often don't make use of sets to their fullest. SDL provides a very powerful set abstraction; one that can deal with any kind of atomic type.

Let's look at an example that works with a set of random numbers:

```
Function RandomNumber(ptr : Pointer) : DObject;
Begin
  Result := Make([Random(1000)]);
End;
Procedure SetStuff;
Var s : DSet;
    I, x : Integer;
Begin
  S := DSet.Create;
  Generate(s, 40, MakeGenerator(RandomNumber));

  For I := 1 to 50 do
    Begin
      X := Random(1000);
      If set.includes([x]) then
        Writeln(x, ' is in the set')
      Else
        Writeln(x, ' is NOT in the set');
    End;
End;
```

This is an example of a very basic set usage. It builds a set full of random numbers, then uses that set to determine if other random numbers are in the set. Let's do something a little

more sophisticated. We know that Delphi supports certain set operations, like set intersection and set unions. SDL supports these as well. Here's an example:

```
Procedure SetOps;
Var s1, s2 : DSet;
    A : DArray;
    Iter : DIterator;
Begin
  S1 := DSet.Create;
  S2 := DSet.Create;
  A := DArray.Create;
  Generate(s1, 100, makeGenerator(RandomNumber));
  Generate(s2, 100, makeGenerator(RandomNumber));
  SetIntersection(s1, s2, a.finish);
  Iter := a.start;
  While iterateOver(iter) do
    Writeln(getInteger(iter), ' is in both sets.');
  FreeAll([s1, s2, a]);
End;
```

This examples generates two sets full of random numbers, then computes the intersection between the two sets. It then prints out the intersection set, which is the set of numbers that are in both sets. We can easily modify this to generate the union of the two sets, which is the set of numbers that are in both sets:

```
Procedure SetOps;
Var s1, s2 : DSet;
    A : DArray;
    Iter : DIterator;
Begin
  S1 := DSet.Create;
  S2 := DSet.Create;
  A := DArray.Create;
  Generate(s1, 100, makeGenerator(RandomNumber));
  Generate(s2, 100, makeGenerator(RandomNumber));
  SetUnion(s1, s2, a.finish);
  Iter := a.start;
  While iterateOver(iter) do
    Writeln(getInteger(iter), ' is in one of the sets.');
  FreeAll([s1, s2, a]);
End;
```

Note that to accomplish this, we only needed to change the setIntersection to a setUnion call.

Lesson 7 – Using the Sort Algorithms

SDL provides two different bases for sorting – the sort and stableSort algorithms. Sort is a quicksort, and stableSort is a merge sort. StableSort has the additional property that, for any items that are equal, their order will be retained after the sort. On very large sorts, stableSort can be faster than quickSort, at the expense of using more memory.

```
Function NameComparator(ptr : Pointer; const obj1, obj2 : DObject) : Integer;
Begin
  Result := CompareText(
      TEmployee( getObject(obj1)).name, TEmployee( getObject(obj2)).name));
```

```
End;
Function BenefitsComparator(ptr : Pointer; const obj1, obj2 : DObject) : Integer;
Var e1, e2 : Boolean;
Begin
  E1 := TEmployee(getObject(obj1)).benefits;
  E2 := TEmployee(getObject(obj2)).benefits;
  If e1 = e2 then
    Result := 0
   Else if e1 then
     Result := -1
   Else
     Result := 1;
End;
Procedure SortDemo;
var a1, a2 : DArray;
begin
  a1 := DArray.Create;
  Generate(1, 10, MakeGenerator(GenEmployee));
  sortWith(a1, MakeComparator(NameComparator));
  PrintContainer(a1);
  a2 := a1.clone as DArray;
  sortWith(a1, MakeComparator(BenefitsComparator));
  stablesortWith(a2, MakeComparator(BenefitsComparator));
  PrintContainers([a1, a2]);
  ObjFree(a1);
  a1.free;
  a2.free;
end;
```

This rather long example shows the essential difference between the two sorting algorithms. We first sort by employee id and print the employees. We then sort with both the sort algorithm and the stableSort algorithm on the benefits field. When we print out a1 (done with sort), the employees are no longer in employee id order, because sort scrambles them up. When we print out a2, we notice that all the employees without benefits (benefits = false) show up first, still in employee id order, followed by those who have benefits. This is the advantage of stable-sorting. You can sort multiple times, and the order is retained (without violating sort concerns).

Lesson 8 – Changing Data Structures

We've mentioned several times that you can change data structures fairly easily with SDL. Let's show how this is possible.

```
Procedure PrintNumber(ptr : Pointer; const obj : DObject);
Begin
  Writeln(getInteger(obj));
End;

Procedure UsingArray;
Var con : DContainer;
    I : Integer;
Begin
  con := DArray.Create;
  Generate(con, 100, MakeGenerator(RandomNumber));

  For I := 1 to 10 do
```

```
    con.remove([Random(100)]);

  ForEach(con, MakeApply(PrintNumber));

 Con.free;
End;

Procedure UsingList;
Begin
Var con : DContainer;
    I : Integer;
Begin
  con := DList.Create;
  Generate(con, 100, MakeGenerator(RandomNumber));

  For I := 1 to 10 do
    con.remove([Random(100)]);

  ForEach(con, MakeApply(PrintNumber));

 Con.free;
End;

Procedure UsingSet;
Var con : DContainer;
    I : Integer;
Begin
  con := DSet.Create;
  Generate(con, 100, MakeGenerator(RandomNumber));

  For I := 1 to 10 do
    con.remove([Random(100)]);

  ForEach(con, MakeApply(PrintNumber));

 Con.free;
End;
```

Note how similar these three examples are – in fact, they're identical except for the container construction call. The code that operates on them is identical. This example is important because it goes to the heart of why you would choose one data structure over another. Let's follow the thought process through these three examples.

Our first example uses arrays. Arrays are good at iteration, and very good for random access, but not so good for addition and removal. In our example we are adding and removing, but not doing any indexed access. This routine is not performing very well, so we might as well try to make it more efficient.

Our second example uses a list. Lists are good at iteration, and very good at insertion and deletion at any point. They are not particularly good at finding items. This second routine performs better because it can quickly add and do the removal operation, but we find that remove runs slowly because list must scan through all of its elements to find the element that needs to remove.

Since we really want that remove to run fast, we change our data structure in the third example to use a DSet. Now every operation runs quickly. Of course, set is not the ideal structure for every situation. Its drawbacks include larger storage requirements, and increased time (log N) for both addition and deletion. The advanced is that instead of a O(N) time for the removal, we have a O(Log N).

Lesson 9 – Transforming Objects

One of the most powerful algorithm sets in SDL is the transform family. Transform will iterate over one or two containers (in its unary and binary forms) and call a specified function with items from each container. The result of the function is then stored at a destination. Let's say that we wanted to create a routine that would fill an array with the hash codes of all our employee names. Here's how we would do it (with transformUnary):

```
Function HashName(ptr : Pointer; const obj : DObject) : DObject;
Begin
  Result := make([JenkinsHashString(TEmployee(getObject(obj)).name)]);
End;
Procedure showTransformUnary;
Var employees, hashcodes : DArray;
Begin
  Employees := DArray.Create;
  Hashcodes := DArray.Create;
  Generate(employees, 20, MakeGenerator(GenEmployee));
  TransformUnary(employees, hashcodes, MakeUnary(HashName));
  FreeAll([employees, hashcodes]);
End;
```

The transform unary algorithm does all the hard work of organizing values and putting them into the output container automatically. Let's try another scenario, in which we want to fill an array with a sum of the hash code and the employee id:

```
Function SumCodes(ptr : Pointer; const obj1, obj2 : DObject) : DObject;
Begin
  Result := Make([
    TEmployee(getObject(obj1)).id + asInteger(obj2) ]);
End;
Procedure showTransformBinary;
Var employees, hashcodes, sums : DArray;
Begin
  Employees := DArray.Create;
  Hashcodes := DArray.Create;
  Sums := DArray.Create;
  Generate(employees, 20, MakeGenerator(GenEmployee));
  TransformUnary(employees, hashcodes, MakeUnary(HashName));
  TransformBinary(employees, hashcodes, sums, MakeBinary(SumCodes));
  FreeAll([employees, hashcodes, sums]);
End;
```

Using the transform algorithms effectively can make your code very small, and very readable.

Lesson 10 – Filtering Objects

This lesson demonstrates another family of SDL functions – the filtering functions. Filtering out a set of objects happens all the time, and SDL is here to help. Our scenario this

time is that we want to examine all of our employees and create a list of those making over $50,000 a year. We then want to sort the list, and print out their names. SDL's filtering and other algorithms make this very easy to accomplish:

```
Function IsRich(ptr : Pointer; const obj : DObject) : Boolean;
Begin
   Result := TEmployee(getObject(obj)).salary >= 50000;
End;
Function NameComparator(ptr : Pointer; const obj1, obj2 : DObject) : Integer;
Begin
  Result := CompareText(
      TEmployee( getObject(obj1)).name, TEmployee( getObject(obj2)).name));
End;
Procedure PrintEmployee(ptr : Pointer; const obj : DObject);
Begin
  With asObject(obj) as TEmployee do
    Writeln('Salary for ', name, ' is ', salary);
End;
Procedure FilterDemo;
Var employees, richGuys : DArray;
    Iter : DIterator;
Begin
  Employees = DArray.CreateWith(MakeComparator(NameComparator));
  RichGuys := DArray.CreateWith(MakeComparator(NameComparator));
  Generate(employees, 50, MakeGenerator(GenEmployee));
  Filter(employees, RichGuys, MakeTest(IsRich));
  Sort(RichGuys);
  ForEach(RichGuys, MakeApply(PrintEmployee));
  ObjFree(employees);
  FreeAll([Employees, RichGuys]);
End;
```

There's one special trick being used here. We making our arrays with CreateWith; that tells SDL about the comparator we want to use with for the container being created. All algorithms will then use that comparator as the default comparator.

Much of the time you can ignore comparators, because SDL puts a fairly intelligent default comparator on your containers. This comparator can sort on any of the atomic types. If you want to have special ordering behavior, such as sorting on a field of an object, you need to provide SDL with a comparator that can do the job. NameComparator in the example above does just that.

Containers

SDL provides the programmer with 11 basic data structures, which cover a large range of programmer's needs. The data structures have good characteristics: efficient implementation, consistent naming, and compatibility with generic algorithms. In addition, SDL's data structures provide a seamless compatibility with Delphi's fundamental data types, as well as its object model.

SDL stores *items* in its data structures, which are descended from **DContainer**. Items are either Delphi primitive data types, or Delphi objects. Therefore, items can be any of the following:

Integer, Boolean, Char, Extended, ShortString (old-style string), Pointer, PChar, Object, Class, WideChar, PWideChar, String (long string), Currency, Interface, WideString.

SDL always stores items by value. *This is very important! SDL does not* own *objects that are inside of its containers. When you put an integer or a string into a container, the value is copied into the container. When you put an object (pointer to object) into a container, the pointer is copied, but the object is not; this is because SDL is storing the pointer, not the object itself.*

When an SDL container is destroyed, it does *not* free the objects that are inside of it. You can use the ObjFree generic algorithm to do this, if you want to.

Because Delphi provides limited language support for certain constructs that would have made creating SDL easier, it is important that you understand exactly *how* SDL stores your items.

All About DObjects

Delphi provides us with a parameter type known as *array of const*. You can pass just about any atomic type or object as part of an array of const. The receiving procedure sees the array of const as an array of TVarRec objects. DObject (SDL's atomic type) is defined precisely the same way as TVarRec. When you add items to a container, you will usually use the add([item]) form. Delphi converts this into an array of TVarRec records, and passes them to the add procedure. SDL *makes copies* of all the items passed in (creates new TVarRec/DObject records for them), and adds them to the container.

If you are just using the helper functions (putInteger, getInteger, atAsInteger, etc.) you don't need to worry about this value copying – it will happen automatically. Periodically, for performance reasons, you may want to interact directly with the DObject records that SDL is storing. If you do this, you need to be aware of the rules.

DObjects can be copied directly, by assigning them to one another. If you do this, you need to ensure that *only one* of the objects is cleaned up with the ClearDObject function. If you want to make a copy, use the CopyDObject function. If you want to ensure that a DObject is empty without clearing it, use InitDObject. SDL uses these functions internally to ensure that all items are copied around cleanly, and that no memory is leaked.

If you retrieve a DObject from a container directly, you need to clean it up when you are finished with it by calling ClearDObject. This is because SDL has created a copy of the DObject and passed it back to you. ClearDObject doesn't do anything for most types, but for Strings, ShortStrings, and Extended values it cleans up associated memory.

If you retrieve a pointer to a DObject (PDObject), you should *not* clean it up. If you examine the SDL source code, you will see the SDL's algorithms rely extensively on retrieving pointers to DObjects. They do this for performance reasons, and also because they wish to manipulate the items in the containers without actually knowing what the types of those items are.

SDL provides two versions of many functions that operate on containers – the first is conventionally named (like *add*), and the second is the direct DObject form, prefixed by an underscore (*_add*). Using the conventional form, you can pass any items in that Delphi will permit in an array of const, which is just about all atomic types and object pointers. You may periodically use the second form when you have a DObject already, which can sometimes happen when you are coding for extreme efficiency. The conventional forms calls the DObject form internally, and automatically.

Another side effect of this is that SDL takes advantage of the array of const feature to allow multiple items to be specified in calls. For example, add([25, 26, 27, 31]) will add all four numbers to its container. SDL will internally loop through each item in the array and add it automatically. This can be a very convenient shortcut at times.

Here is a list of functions that operate directly on DObjects:

```
procedure SetDObject(var obj : DObject; value : array of const);
procedure InitDObject(var obj : DObject);
procedure CopyDObject(const source : DObject; var dest : DObject);
procedure MoveDObject(var source, dest : DObject);
procedure ClearDObject(var obj : DObject);
```

Example Code

SDL comes with many examples. Please examine the **SDLExamples.pas** file that came with your distribution – it's a great guide to the usage of various SDL features and containers. It also demonstrates many "correct" ways to use SDL.

Container hierarchy

SDL divides its containers into a simple hierarchy. Moving down the hierarchy increases the functionality available in each container. This has advantages; note that if you know that a given situation requires a mapping structure, you can assume in your code that the data structure is a descendent of DAssociative. Then, later, you can use any of the subclasses of DAssociative to store the actual data, and none of your code that uses the data will need to change. You can further insulate yourself by making sure that you use iterators wherever possible.

Figure 1 shows the SDL container class hierarchy:

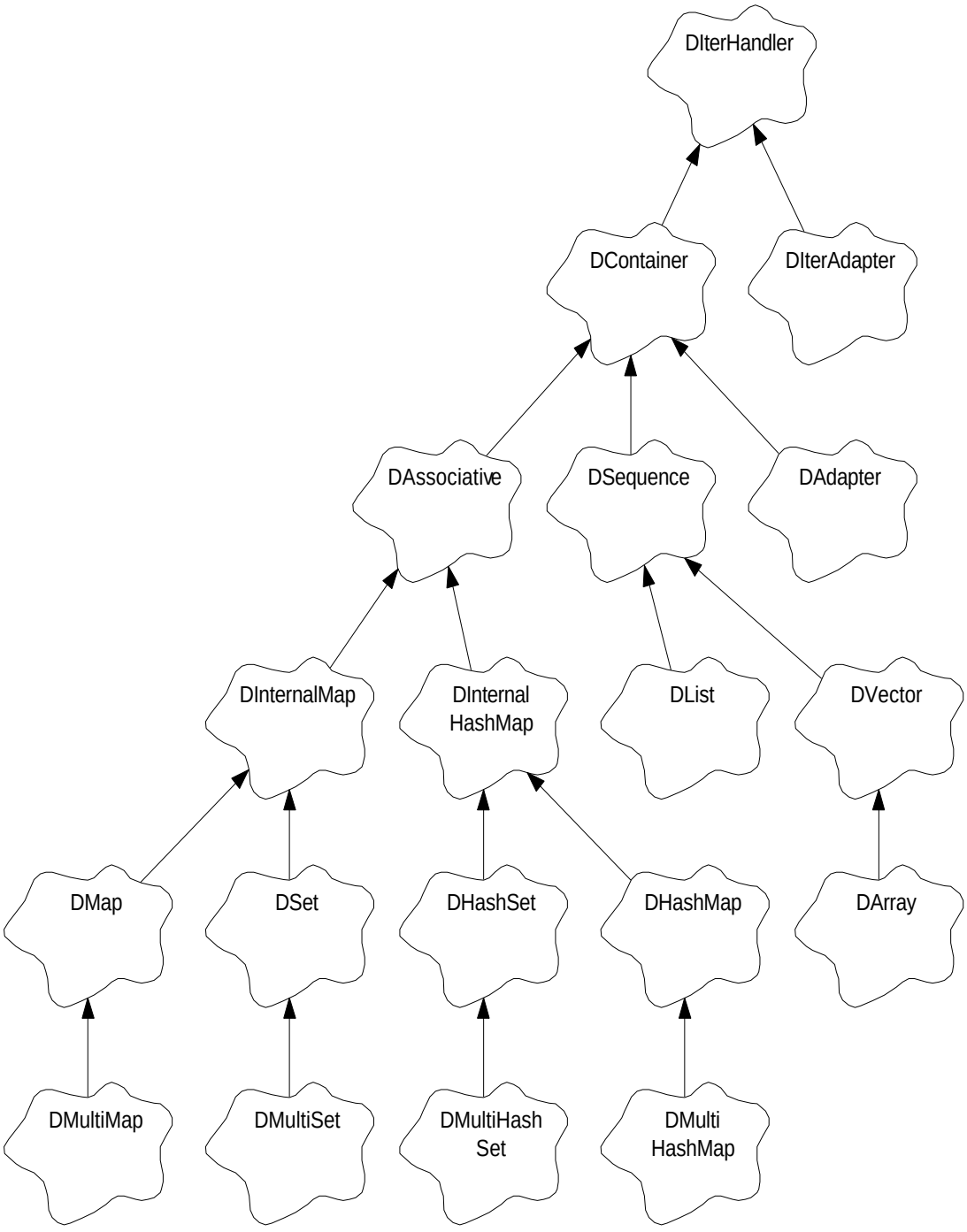

Figure 1- SDL Class Hierarchy

DIterator

Iterators (**DIterator**) are absolutely fundamental to working with SDL. Generic algorithms (and very likely your algorithms) operate by manipulating and using iterators, rather than working with the container classes directly. All containers provide methods for retrieving

their starting and finishing iterators. Once you have an iterator, you can use a set of global functions to move them from item to item, retrieve the item the iterator is positioned at, and put new items at the iterator position. All of these operations on iterators are independent of the containers underneath them. Here is an example of retrieving and using an iterator:

```
Procedure test(c : DContainer);
var iter : DIterator;
begin
  iter := c.start;
  while not atEnd(iter) do
    begin
      writeln(getInteger(iter));
      advance(iter);
    end;
end;
```

This example accepts any kind of container, then prints out each item in the container, assuming that item is an integer. Note the use of the atEnd function, which tests to see if an iterator is positioned after the last item in the container. When the iterator is at the end of a container, you cannot read from it (with a getXXX function). Some containers do permit you to write to an iterator that is positioned at the end, but not all (associative containers do not support this). SDL provides a full set of getXXX functions; one is provided for each atomic type. There are equivalent putXXX functions as well. Here's the list:

```
function getInteger(const iterator : DIterator) : Integer;
function getBoolean(const iterator : DIterator) : Boolean;
function getChar(const iterator : DIterator) : Char;
function getExtended(const iterator : DIterator) : Extended;
function getShortString(const iterator : DIterator) : ShortString;
function getPointer(const iterator : DIterator) : Pointer;
function getPChar(const iterator : DIterator) : PChar;
function getObject(const iterator : DIterator) : TObject;
function getClass(const iterator : DIterator) : TClass;
function getWideChar(const iterator : DIterator) : WideChar;
function getPWideChar(const iterator : DIterator) : PWideChar;
function getString(const iterator : DIterator) : String;
function getCurrency(const iterator : DIterator) : Currency;
function getVariant(const iterator : DIterator) : Variant;
function getInterface(const iterator : DIterator) : Pointer;
function getWideString(const iterator : DIterator) : WideString;
```

Here is the equivalent list of putXXX functions:

```
procedure put(const iterator : DIterator; objs : array of const);
procedure putInteger(const iterator : DIterator; value : Integer);
procedure putBoolean(const iterator : DIterator; value : Boolean);
procedure putChar(const iterator : DIterator; value : Char);
procedure putExtended(const iterator : DIterator; const value : Extended);
procedure putShortString(const iterator : DIterator; const value : ShortString);
procedure putPointer(const iterator : DIterator; value : Pointer);
procedure putPChar(const iterator : DIterator; value : PChar);
procedure putObject(const iterator : DIterator; value : TObject);
procedure putClass(const iterator : DIterator; value : TClass);
procedure putWideChar(const iterator : DIterator; value : WideChar);
procedure putPWideChar(const iterator : DIterator; value : PWideChar);
procedure putString(const iterator : DIterator; const value : String);
```

```
procedure putCurrency(const iterator : DIterator; value : Currency);
procedure putVariant(const iterator : DIterator; const value : Variant);
procedure putInterface(const iterator : DIterator; value : Pointer);
procedure putWideString(const iterator : DIterator; const value : WideString);
```

There is also the output function, which combines the writing of a value to an iterator's position with advancing the iterator:

```
procedure output(var iterator : DIterator; objs : array of const);
```

Making use of these functions lets you get your atomic values in and out of DObjects easily and quickly. SDL has a great number of these type conversion functions – make use of them! SDL has these details done right, so you don't have to code them yourself.

Certain containers store pairs (DMap, DMultiMap, DHashMap, DMultiHashMap). When you retrieve an iterator from one of these containers, the iterator will walk over the *values* in the key-value pairs. If you want to examine the keys, call *SetToKey(iterator)*. After making that call, the getXXX functions will return the key part of the pair. To retrieve values, call *SetToValue(iterator)*.

DIterators are records. They have been specifically designed to ensure that you can copy them freely, return them as results from functions, and assign them between variables. Each DIterator contains enough information to indicate which container it came from, the position it has in that container, and certain flags indicating the status of the iterator. DIterators are exactly 16 bytes in length, and will stay that way if possible (to accelerate reading and writing DIterators are one *cache line* in length).

DIterators can be grouped into classes. Each class adds more functionality. The class of an iterator is dependent on the data structure that produced it. Certain data structures only support very simple operations; this means that their iterators are simple. Other data structures provide much fuller iterator operation support. What follows is a description of the iterator classes.

Forward Iterators

The simplest iterator is one that can only move forward. You can retrieve the iterator from the container (usually with the start function), and you can move it forward (with *advance*) until it is at the end of the container. You can use the following functions with forward iterators:

```
advance - move an iterator to the next item
getXXXX - retrieve the item the iterator is positioned at
equals - test if two iterators are at the same position
putXXXX - store an item where the iterator is positioned
output – store an item where the iterator is positioned, and advance the iterator
advanceBy - advance the iterator multiple positions (retreat if negative)
atStart - tests to see if an iterator is at the start of a container
atEnd - test to see if an iterator is at the end of a container
getContainer - retrieves the container associated with an iterator
distance - determines the number of positions between two iterators
```

Bidirectional Iterators

Bidirectional iterators extend forward iterators so that they can move backwards. The following functions work only with bidirectional (and better) iterators:

```
retreat - moves an iterator backwards to the previous item
retreatBy - moves an iterators backwards by a number of positions
getAt - retrieves the item at a certain position relative to the iterator
putAt - stores an item relative to the position of the iterator
```

Random Iterators

Random iterators extend bidirectional iterators to implement efficient movement of the iterator to a random position in the container, usually indicated by an integer. The following functions work with random iterators:

```
index - determines the current position of the iterator, as an integer
less - determines if one iterator is pointing "earlier" in a container
```

Iterator Adapters

Iterators can be wrapped with adapters to provide additional or different behavior. SDL has one such adapter DIterSkipper, which alters the behavior of the iterator it is attached to so that advances and retreats move by a certain number of positions.

You can create your own adapters by subclassing DIterAdapter.

Iterator adapters work by substituting themselves into the iterator's Handler field. All functions that are executed on the iterator are routed through the Handler. The adapter can then pass the request unmodified to the original handler (which is often the container that produced the iterator), or they can modify the request, or do any other processing that is required.

DContainer

DContainer is the base class for all container classes in the SDL library. It defines a basic set of public operations that all containers support, defines essential *comparator* functionality, and anchors the container hierarchy with a set of abstract, virtual operations that concrete container classes must implement. It also provides default implementations of a number of basic operations – these default implementations are very "lowest common denominator". Subclasses may decide to implement these operations in a more efficient way; to do so, they simply override the function with the new, improved, and more efficient version. Here is a list of the standard methods provided by containers:

```
add - add items to containers
clear - clears all items from a container
lone - creates a shallow copy of a container
contains - determines if a container has a given item in it
count - counts the number of times a given item occurs in a container
finish - returns an iterator positioned after the last item in the container
getComparator - returns the comparator currently being used by the container
isEmpty - determines if the container has any items in it
maxSize - returns the largest number of items the container can hold
remove - erases (removes) matching items from a container
```

```
size - returns the number of items in a container
start - returns an iterator positioned on the first item in the container
```

Since all containers provide these functions, quite a bit can be done without knowing anything about the details of a data structure being used. The DContainer interface, coupled with the iterator manipulation functions, constitute a powerful abstraction of data structures away from your code.

Comparators

Comparators are the functions used by SDL containers to compare elements. For certain structures, such as maps, comparators are absolutely integral to the function of the container, as they provide the mechanism by which items are compared to one another. Other container classes, like arrays, don't use comparators to the same extent, but they are still present.

A comparator is defined as follows:

```
DComparator = function (const obj1, obj2 : DObject) : Integer of object;
```

An alternate form of comparators is defined like this:

```
DComparatorProc = function(ptr : Pointer; const obj1, obj2 : DObject) : Integer;
```

These two definitions are compatible with each other because they take advantage of a trick – calls on procedures of object always have *self* as the first parameter. By placing a dummy pointer in this position, we can make our comparators either *closures* (procedures on objects) or functions (standalone functions).

If you are defining your own comparators, you should always ensure that the function returns **less than zero** if obj1 is less than obj2, **zero** if obj2 equals obj2, and **greater than zero** if obj2 is greater than obj1.

You will be using this most frequently when you are comparing two TObject descendents, which will be contained inside the DObjects. You can access the objects by using **asObject**, like so:

```
ACar := asObject(obj1) as TCar
```

Comparators can be called very frequently, so you will want to try and make them as efficient as possible. Note that a little more knowledge about how DObjects are formed can help in this efficiency:

```
Result := TCar(obj2.VObject)^.FValue - TCar(obj1.VObject)^.FValue;
```

This comparator compares two TCar objects based on the their FValue fields. Note the trick of subtracting the first from the second – this will give us a result with the correct sign for the result.

Constructing Containers

All containers support two forms of construction, and also have the ability to clone themselves. Here are the two forms:

```
Constructor Create; virtual;
Constructor CreateWith(comparator : DComparator); virtual;
```

Note that these constructors are virtual – this is very handy for algorithms that need to create auxiliary storage and still want to work independently of container type. The plain *Create* constructor makes a container that uses the standard comparator routine. This default comparator routine knows how to compare all atomic Delphi types, such as strings, integers, floats, and so forth. It compares objects by comparing their pointers; this is adequate for things like set membership, but is inadequate for locating items or any kind of ordering. If you're storing objects you'll probably want to provide your own comparator that works on one of the fields of the object (see the section above for an example).

Containers can also clone themselves:

```
Function clone : DContainer; virtual;
```

This functions creates a complete new copy of the container, with copies of all items inside the container. Note that if the container had TObjects inside of it, the pointers will be copied but not the objects themselves (a *shallow* copy).

Certain container classes provide additional constructor functions that are appropriate to their specific data structures. For example, the DArray class provides a CreateSize constructor, which makes room for a certain number of items in the array.

Number of Items

Every container responds to the *size* function, which returns the number of items in the container. Note that you should *not* use the size function to iterate over containers. If you do, you'll limit yourself to those containers that have random access iterators! Instead, retrieve an iterator with the *start* function and *advance* it until *atEnd* returns true.

MaxSize returns the largest number of items that you can place in a given container type. Note that the number returned assumes you have unlimited memory; it is more of a theoretical limit than a hard limit.

Contains determines if the container has a certain item inside of it. *Count* iterates through the container and determines the number of items that match the object specified.

Adding Items

Containers almost always support the *add* function, which puts new items into the container (notable exceptions are the Map classes, which only accept pairs of items, in key – value form). There are often other functions that add items to the container, but most of them are data structure specific. Use the simple *add* call whenever you can – it'll make your code as independent of the underlying data structure as possible.

Removing Items

You can remove an item by calling *remove*. Remove operates on a value-oriented basis. The container uses its comparator to determine if an item needs to be removed. Remove will generally only remove one item. There are other forms of the call (RemoveN) that can be used to remove more than one item that matches the value passed in.

If you have an iterator positioned at an item, you can use the *removeAt* function. This will erase the element the iterator is over. It also invalidates the iterator.

To remove all items from a container, use the *clear* function. All containers support clear. Note that calling clear does *not* free any TObjects that the container might be holding pointers to. You can call the *FreeAll* algorithm to destroy the objects before clearing the container.

Various subclasses have additional operations for removing elements that operate in ways specific to that data structure.

Retrieving Items

Container-independent retrieval and iteration is achieved by using iterators. All containers support *start* and *finish*. Calling start retrieves an iterator positioned at the first item in the container. Finish returns an iterator positioned just after the last element – this non-existent position is known as the finish position. If the container is empty, the start function may return an iterator that is in the finish position. You can test whether an iterator is at the finish position with the following function:

```
Function AtEnd(const iter : DIterator) : Boolean;
```

AtEnd is often used in a construct like the following:

```
Procedure test(con : DContainer);
Var iter : DIterator;
    I : Integer;
Begin
  iter := con.start;
  while not atEnd(iter) do
  begin
    <.. do something ..>
    I := getInteger(iter);
    writeln(I);
    advance(iter);
  end;
end;
```

The atEnd procedure invokes a data structure-dependent method of determining if the iterator is positioned at the finish of the container.

DSequence

DSequence is a container that holds its items in…a sequence! Items placed in a sequence-derived container will be retrieved in the order that they were added. DSequences maintain their ordering. Note that a DSequence is not necessarily indexed (with an integer). Double-linked lists are DSequence-derived. Double-linked lists offer rapid insertion and deletion at any point.

Some of the functions available on DSequences are index-based. These functions are not necessarily efficiently implemented by certain kinds of DSequences. Index-based functions are generally only efficiently performed on DVector-based containers, but it will vary. DDeques provide an intermediary type of container that performs well under many conditions.

Adding Items

There are a number of additional functions for adding items that DSequence provides:

```
PutAt(pos, item)
PushBack(item)
PushFront(item)
```

Retrieving Items

DSequence provides the following additional methods for retrieving items from sequence-type containers:

```
At(pos)
AtAsXXXX(pos)
Back
Front
IndexOf(item)
PopFront
PopBack
```

Removing Items

To remove items either use *remove* or use *removeAt*, which removes the item an iterator is positioned at. Be aware that removing the item pointed to by an iterator will usually invalidate that iterator.

DVector

A DVector is a DSequence for which each item can be addressed by an integer index. DArrays and DDeques are DVectors. DVectors are frequently slower to add and delete from in the middle of the structure, but offer very rapid access to individual elements through an index.

These are the additional functions available with DVectors:

```
function capacity : Integer;
procedure ensureCapacity(amount : Integer);
procedure insertAtIter(iterator : DIterator; objs : array of const);
```

```
procedure insertAt(index : Integer; objs : array of const);
procedure insertMultipleAtIter(iterator : DIterator; count : Integer; objs : array of
const);
procedure insertMultipleAt(index : Integer; count : Integer; objs : array of const);
procedure insertRangeAtIter(iterator : DIterator; _start, _finish : DIterator);
procedure insertRangeAt(index : Integer; _start, _finish : DIterator);
procedure removeAt(index : Integer);
procedure setCapacity(amount : Integer);
procedure trimToSize;
```

DAssociative

The DAssociative classes place significantly more organization on their contents than do the other classes. The structure of the data is directly determined by the values that are placed inside of them. There are two major families of associative classes : Hash-based and red-black tree-based. Hash based structures are appropriate where comparisons are slow, or there are smaller numbers of items, and where memory is not as important. Red-black structures are appropriate where ensuring access time is highly important, as red black trees are balanced data structures. Red-black trees have a guaranteed upper bound on the amount of time it takes to execute their various operations.

Sets and Maps

Associatives are divided into two types – sets and maps. They are, in fact, implemented exactly the same way (they both store pairs). Sets usually contain a null value in the second half of the pair, and all their operations work on the key, by default.

Maps store *pairs* – they associative a given value with a given key. They are exceptionally useful data structures – in fact, it's estimated that 90% of all container usage in programs is map-based, where efficient and easy to use map implementations are available. SDL provides four different map structures: DMap, DMultiMap, DHashMap, DMultiHashMap. DMap and DMultiMap are red-black tree based, and DHashMap and DMultiHashMap are hash-based.

The *multi* designator indicates whether or not the container will accept multiple values for the same key. It is often desirable to have a container store only one value for a given key, and if another value is set to the same key, it replaces the first value. For these situations, do *not* use the multi versions.

Multi-maps will allow any number of pairs with the same key to be added to the map.

Where maps associate a key with a value, sets are concerned only with the key. For sets, the value **is** the key. Other than that, they generally perform exactly the same way that maps do.

Adding Elements

To add elements to a set, use *add*. To add elements to a map, use *putPair* or *putAt*. Each type of container will ensure that you use the correct form with assertions. Note that is doesn't make any sense to try to add elements directly to a map (because you haven't

supplied the *value* part of the pair), and it doesn't make any sense to add pairs to a set (because there isn't any value).

Finding Elements

To find if a key is in a map, use *locate*. To find if a value is in a set, you can also use *locate*. For maps, locate returns an iterator positioned at the first item (value) that matches the key. For sets, the iterator is positioned at the key itself (which is also the data!).

Note that if you want to retrieve a value from a map or set and you don't know if the value is actually there, use locate. Test the iterator that locate returns – if it's *atEnd*, then the key doesn't exist in the map, and you'll have to add it.

Removing Elements

You can remove elements from maps using *remove*.

Container Adapters

Creating Your Own Containers

Creating your own containers is not too difficult – basically you need to override a series of virtual functions that DContainer has defined. Some of them don't need to overridden – there are default implementations. Those default implementations may not be the fastest way to perform an operation on your new data structure, so you may want to implement a custom version.

You will very likely need the SDL source code to implement your own containers, particularly if you need to change the definition of DIterator in any way.

Frequently Asked Questions

How do I get the number of items in a container?

The *size* function returns the number of items in any container.

How do I add items to a container?

If you're adding to a non-map container, use the *add* function:

```
Container.add([value]);
```

If you're adding to a map-based container, use the *putAt* or *putPair* functions:

```
Container.putAt(['testing', 'again'], [1, 2]);
Container.putPair(['toast', 10]);
```

How do I iterate over a container?

Declare an iterator, then call the container's *start* function to retrieve an iterator for the container. Loop until the iterator is at the end. Here are the two basic techniques for doing this:

```
Procedure Example(con : DContainer);
Var iter : DIterator;
Begin
   Iter := con.start;
   While not atEnd(iter) do
     Begin
        Advance(iter);
     End;

   Iter := con.start;
   While iterateOver(iter) do
     Begin
       // no advancement necessary – but don't reuse the iterator once the loop is
done!
     End;
end;
```

How do I retrieve the keys from a map container?

Call SetToKey on your iterator, then retrieve values with the getXXX functions. Call SetToValue to retrieve the value part of the key-value pair again.

How do I sort a sequence?

Use the sort or stableSort algorithms.

```
Sort(container);
```

Note that sorting only makes sense on sequential containers. Sorting an associative structure will result in exceptions.

Why does SDL use functions instead of class members for its algorithms, and for iterator operations?

There are two reasons. First, SDL iterators can operate on any class that implements the DIterHandler interface. SDL's containers are descended from DContainer, which inherits from DIterHandler, but other containers or container-like classes don't have to be.

The second reason is for compatibility and interoperability with STL and JGL. Both of those packages use the functional style of programming. In STL, this was done to enable all algorithms to operate on C-style arrays as well as containers. Soletta assumes that JGL was coded this way to maintain compatibility with STL.

Another effect of this is that algorithms are very cleanly separated from the container code.

How do I find items in a map?

Call the *locate* function, and test the returned iterator:

```
Iter := map.locate([value]);
If not atEnd(iter) then
  Writeln('Found it: ', getInteger(iter));
```

Algorithms

SDL contains a large number of *generic algorithms*. These algorithms are solutions to problems that present themselves over and over again while you're coding. Study of this section is very important to getting the maximum benefit out of SDL. What you need to learn to do is *recognize when a common problem occurs, then substitute the appropriate generic algorithm.*

As a very simple example, we all know that a need to sort objects occurs all the time. Rather than coding your own sort, you can call either **sort** or **stableSort** in the SDL library. You're probably used to calling a sort procedure in a library, rather than creating your own.

Let's look at another situation: Let's say you need have a bunch of employee objects and you need to cull out the ones who are waiting for expenses to be reimbursed. The *removeIf* algorithm can do this for you. Let's say that you want to create reimbursement objects for those employees that need to be paid. The *transformUnary* algorithm is ideal for this case.

A Note About Ranges

Algorithms that accept a range (_start to _end, typically) do *not* apply themselves to the _end element. They stop at the position before the _end supplied. This is done so that you can conveniently pass (container.start, container.finish) as arguments to an algorithm. It is generally illegal to do anything at *container.finish.* Certain containers *may* permit addition or writing at this location, but not all.

Naming Conventions

Because Delphi 3 does not support overloading, SDL uses a naming convention for its functions to achieve the same thing. For each algorithm there are often several ways to call it, depending on what you want to achieve. Decorators are added to the algorithm name to arrive at the right call mix. The following decorators are used:

- In – Perform the algorithm in a certain range (usually a _start - _end pair).
- To – Send the output of the algorithm to a destination (usually an _output iterator).
- If – Use a test to determine if the algorithm should operate on that element (usually a DTest).
- With – A comparator is provided that should be used in place of the container's comparator (a DComparator is passed to the algorithm).

So, for example, the routine UniqueInWithTo performs the *unique* algorithm, *in* a range, *with* a comparator, *to* a destination. It's defined like this:

```
procedure uniqueInWithTo(_start, _end, dest : DIterator; compare : DBinaryTest);
```

These rules do tend to vary by algorithm, because each algorithm has a certain set of parameters that must be provided to it. The naming convention is followed wherever possible, though.

Applying

forEach

```
procedure forEach(container : DContainer; unary : DApply);
procedure forEachIn(_start, _end : DIterator; unary : DApply);
procedure forEachIf(container : DContainer; unary : DApply; test : DTest);
procedure forEachInIf(_start, _end : DIterator; unary : DApply; test : DTest);
```

Applies a unary function to each element in a container. Frequently you'll need to pass each item in a container to a function – perhaps you are printing, or summing the values in the container, or you need to perform some kind of special processing on each item. The various forms of the forEach function can do this for you. Remember that you can convert a non-closure function into the DTest these algorithms require with the MakeTest function.

ForEach applies the unary function to each item in the container.

ForEachIn applies the unary function to each item in the range given (not including the item at the _end position).

ForEachIf applies the test specified to each item in the container. For those items that return true on the test, the unary function is called.

ForEachInIf applied the test to each item in the range. Those items that return true are passed to the unary function.

Inject

```
function _inject(container : DContainer; const obj : DObject; binary : DBinary) :
DObject;
function _injectIn(_start, _end : DIterator; const obj : DObject; binary : DBinary)
: DObject;
function inject(container : DContainer; obj : array of const; binary : DBinary) :
DObject;
function injectIn(_start, _end : DIterator; obj : array of const; binary : DBinary)
: DObject;
```

The inject family moves a calculation's results along through an entire range or container. It is useful if, for example, you want to sum the values in a container. Inject takes a seed value (the obj parameter). It calls binary for each object in the range or container, passing the seed value as the first parameter and the item as the second parameter. The results of the binary function call become the new seed. After all items have been processed, the last result is returned.

Comparing

Equal

```
function equal(con1, con2 : DContainer) : Boolean;
function equalIn(start1, end1, start2 : DIterator) : Boolean;
```

The equal algorithm determines if the two containers or ranges are equal to each other. They are equal to each other if each item in the range equals the corresponding item in the other range, and the ranges (or containers) are of equal length.

LexicographicalCompare

```
function lexicographicalCompare(con1, con2 : DContainer) : Boolean;
function lexicographicalCompareWith(con1, con2 : DContainer; compare : DComparator) :
Boolean;
function lexicographicalCompareIn(start1, end1, start2, end2 : DIterator) : Boolean;
function lexicographicalCompareInWith(start1, end1, start2, end2 : DIterator; compare
: DCOmparator) : Boolean;
```

Lexicographical comparison compares the items in two containers or ranges one by one. The first time a difference is found between two items, it returns less than zero if the item in the first range or container was less than the second, or returns greater than zero if the item in the first range or container was greater than the second. In either case, the comparison stops as soon as a difference is detected.

Median

```
function _median(const obj1, obj2, obj3 : DObject; compare : DComparator) : DObject;
function median(objs : array of const; compare : DComparator) : DObject;
```

Median returns the middle of three values, using the comparator specified. You must pass exactly three values to it.

Mismatch

```
function mismatch(con1, con2 : DContainer) : DIteratorPair;
function mismatchWith(con1, con2 : DContainer; bt : DBinaryTest) : DIteratorPair;
function mismatchIn(start1, end1, start2 : DIterator) : DIteratorPair;
function mismatchInWith(start1, end1, start2 : DIterator; bt : DBinaryTest) :
DIteratorPair;
```

Mismatch determines the point at which two sequences begin to differ. It returns an iterator pair, the first of which is positioned at the position in the first sequence where the difference began, and the second of which is positioned in the second sequence.

Mismatch returns where two containers begin to differ. If no difference is found the first part of the iterator pair is set to con1's atEnd.

MismatchWith returns where two containers begin to differ, using the binary test supplied.

MismatchIn returns where two sequences (ranges, identified by iterators) begin to differ. If no difference is found, the first pair is set to *end1*.

MismatchInWith returns where two sequences begin to differ using the binary test supplied.

Copying

Copy

```
function copyContainer(con1, con2 : DContainer) : DIterator;
function copyTo(con1 : DContainer; iterator : DIterator) : DIterator;
function copyInTo(_start, _end, output : DIterator) : DIterator;
function copyBackward(_start, _end, output : DIterator) : DIterator;
```

CopyContainer Copies the contents of con1 to con2. An iterator is returned that is positioned at the end of con2.

CopyTo copies the contents of con1 to the iterator given. The iterator is advanced with *output*.

CopyInTo copies the elements in the range given to the output iterator.

CopyBackward copies the elements from the range given to the output iterator, in reverse order.

Counting

Count

```
function count(con1 : DContainer; objs : array of const) : Integer;
function countIn(_start, _end : DIterator; objs : array of const) : Integer;
function countIf(con1 : DContainer; test : DTest) : Integer;
function countIfIn(_start, _end : DIterator; test : DTest) : Integer;
```

Count determines the number of items in con1 that are equal to each item passed for objs. If more than one item is passed to objs, the counts are summed.

CountIn counts the number of items in the range _start to _end that are equal to each item passed for objs. If more than one item is given, the counts are summed.

CountIf Determines the number of items in the container that pass the test supplied.

CountIfIn determines the number of items in the range _start to _end that pass the test supplied.

Filling

Fill

```
procedure fill(con : DContainer; obj : array of const);
procedure fillN(con : DContainer; count : Integer; obj : array of const);
procedure fillIn(_start, _end : DIterator; obj : array of const);
```

Fill fills con with the specified value (there must be only one). The currently set size of the container is used to determine how many items are to be put there.

FillN fills con with count copies of a value. If the container isn't large enough, it will have more values added to its end, and will expand to the correct size.

FillIn fill the range specified with the value given.

Generate

```
procedure generate(con : DContainer; count : Integer; gen : DGenerator);
procedure generateIn(_start, _end : DIterator; gen : DGenerator);
procedure generateTo(dest : DIterator; count : Integer; gen : DGenerator);
```

The generate algorithm fill containers or ranges with the output of a given generator function. The goal of the generator function is to create DObjects. The DObjects are stored into the target.

Filtering

Unique

```
Function unique(con : DContainer) : DIterator;
Function uniqueIn(_start, _end : DIterator) : DIterator;
Function uniqueWith(con : DContainer; compare : DBinaryTest) : DIterator;
Function uniqueInWith(_start, _end : DIterator; compare : DBinaryTest) : DIterator;
Function uniqueTo(con : DContainer; dest : DIterator) : DIterator;
Function uniqueInTo(_start, _end, dest : DIterator) : DIterator;
Function uniqueInWithTo(_start, _end, dest : DIterator; compare : DBinaryTest) :
DIterator;
```

Unique ensures that every item in the range or container is unique. If you have the sequence (1,2,3,4,4,5,6,6,7) calling unique on that sequence will result in (1,2,3,4,5,6,7,undefined,undefined). In addition, the algorithm returns an iterator positioned at the first undefined value.

Filter

```
procedure Filter(fromCon, toCon : DContainer; test : DTest);
function FilterTo(con : DContainer; dest : DIterator; test : DTest) : DIterator;
function FilterInTo(_start, _end, dest : DIterator; test : DTest) : DIterator;
```

Filter copies items to a destination if they pass a test. Each item passed to the test – if the test returns true, the item is copied to the output. The filterTo and FilterInTo functions return an iterator positioned after where the last item was written to the destination.

Finding

AdjacentFind

```
function adjacentFind(container : DContainer) : DIterator;
function adjacentFindWith(container : DContainer; compare : DBinaryTest) : DIterator;
function adjacentFindIn(_start, _end : DIterator) : DIterator;
```

```
function adjacentFindInWith(_start, _end : DIterator; compare : DBinaryTest) :
DIterator;
```

AdjacentFind determines if there are two equal, consecutive items in a sequence. It returns an iterator positioned at the first one if it finds two such items. If it doesn't find any, it returns an iterator positioned at the end of the container if given a container, or at the end of the range if given the range.

BinarySearch

```
function binarySearch(con : DContainer; obj : array of const) : DIterator;
function binarySearchIn(_start, _end : DIterator; obj : array of const) : DIterator;
function binarySearchWith(con : DContainer; compare : DComparator; obj : array of
const) : DIterator;
function binarySearchInWith(_start, _end : DIterator; compare : DComparator; obj :
array of const) : DIterator;
```

BinarySearch relies on the fact that the sequence it is given is sorted. It will very efficiently locate an item in a sorted sequence. It returns an iterator positioned at the item.

Detect

```
function detectWith(container : DContainer; compare : DTest) : DIterator;
function detectInWith(_start, _end : DIterator; compare : DTest) : DIterator;
```

Detect locates the first item in a container or range for which the test returns true. It returns an iterator positioned at the end if such an item is not found.

Every

```
function every(container : DContainer; test : DTest) : Boolean;
function everyIn(_start, _end : DIterator; test : DTest) : Boolean;
```

Every determines if the test returns true for every element in the container or range. It does a giant AND of test for every element in the range. It short-circuits, so the first time the test returns false, it will return.

Find

```
function find(container : DContainer; obj : array of const) : DIterator;
function findIn(_start, _end : DIterator; obj : array of const) : DIterator;
function findIf(container : DContainer; test : DTest) : DIterator;
function findIfIn(_start, _end : DIterator; test : DTest) : DIterator;
```

Locate an object in a container, returning an iterator positioned where the object was found. If no object is found, an atEnd iterator is returned. The third and fourth form use a test instead of the container's comparator.

Some

```
function some(container : DContainer; test : DTest) : Boolean;
function someIn(_start, _end : DIterator; test : DTest) : Boolean;
```

Some determines if any of the items in a container return true for the given test. Some short-circuits, so the first item that returns true causes the algorithm to return true.

Freeing and Deleting

ObjFree

```
procedure objFree(container : DContainer);
procedure objFreeIn(_start, _end : DIterator);
```

ObjFree assumes that every item in a container is an object. It calls TObject.Free on each item.

ObjDispose

```
procedure objDispose(container : DContainer);
procedure objDisposeIn(_start, _end : DIterator);
```

ObjDispose assumes that every item in a container or range is a pointer to a heap allocated object (allocated with GemMem); it calls FreeMem on the pointer.

ObjFreeKeys

```
procedure objFreeKeys(assoc : DAssociative);
```

ObjFreeKeys performs the same function as ObjFree, but does it on the keys in the range, not on the values. This is useful if you are have a map that maps objects to some other type.

Hashing

OrderedHash

```
function orderedHash(container : DContainer) : Integer;
function orderedHashIn(_start, _end : DIterator) : Integer;
```

During coding, it is often convenient to convert values, or a range of memory, into a single numeric value that has almost-random characteristics. This can be used to rapidly identify objects, or to sort objects when no other alternatives are available. SDL provides the orderedHash algorithm to create these numeric codes. The ordered hash algorithm has the addition characteristic that the hash code produced will be sensitive to and affected by the order of the items in the container that's being hashed. If this level of sensitivity is not required, use the unorderedHash algorithm, which is slightly more efficient.

UnorderedHash

```
function unorderedHash(container : DContainer) : Integer;
```

```
function unorderedHashIn(_start, _end : DIterator) : Integer;
```

The unorderedHash algorithm is identical to the orderedHash algorithm, except that the hash code produced is not sensitive to the order of the items in the container or range. This is slightly more efficient to calculate than the orderedHash.

Removing

Remove

```
function remove(container : DContainer; objs : array of const) : DIterator;
function removeIn(_start, _end : DIterator; objs : array of const) : DIterator;
function removeTo(container : DContainer; output : DIterator; objs : array of const)
: DIterator;
function removeInTo(_start, _end, output : DIterator; objs : array of const) :
DIterator;
```

Removes all matching items from the container or range it is given. The size of the container doesn't change; the remove family of functions return an iterator positioned at the end of the new sequence.

removeCopy

```
function removeCopy(source, destination : DContainer; objs : array of const) :
DIterator;
function removeCopyTo(source : DContainer; output : DIterator; objs : array of const)
: DIterator;
function removeCopyIn(_start, _end, output : DIterator; objs : array of const) :
DIterator;
function removeCopyIf(source, destination : DContainer; test : DTest) : DIterator;
function removeCopyIfTo(source : DContainer; output : DIterator; test : DTest) :
DIterator;
function removeCopyIfIn(_start, _end, output : DIterator; test : DTest) : DIterator;
```

The removeCopy algorithm copies a sequence of items from one location to another, removing any matching items as it goes.

removeIf

```
function removeIf(container : DContainer; test : DTest) : DIterator;
function removeIfIn(_start, _end : DIterator; test : DTest) : DIterator;
function removeIfTo(container : DContainer; output : DIterator; test : DTest) :
DIterator;
function removeIfInTo(_start, _end, output : DIterator; test : DTest) : DIterator;
```

The removeIf and removeIfIn algorithms remove any items from a sequence for which the test returns true. RemoveIfTo and RemoveIfInTo copy the sequence of items, removing any for which the test returned true.

Replacing

Replace

```
function replace(container : DContainer; objs1, objs2 : array of const) : Integer;
function replaceIn(_start, _end : DIterator; objs1, objs2 : array of const) :
Integer;
```

Replaces all items in the container or sequence that match obj1 with obj2. If you pass more than one object for objs1 and objs2, the algorithms runs multiple times, doing each pair of objects.

ReplaceCopy

```
function replaceCopy(con1, con2 : DContainer; objs1, objs2 : array of const) :
Integer;
function replaceCopyTo(container : DContainer; output : DIterator; objs1, objs2 :
array of const) : Integer;
function replaceCopyInTo(_start, _end, output : DIterator; objs1, objs2 : array of
const) : Integer;
function replaceCopyIf(con1, con2 : DContainer; test : DTest; objs : array of const)
: Integer;
function replaceCopyIfTo(container : DContainer; output : DIterator; test : DTest;
objs : array of const) : Integer;
function replaceCopyIfInTo(_start, _end, output : DIterator; test : DTest; objs :
array of const) : Integer;
```

ReplaceCopy copies a sequence to a new container or iterator, replacing each item that matches obj1 with obj2 as it copies. The IF variants use test to determine if the replacement should happen or not.

ReplaceIf

```
function replaceIf(container : DContainer; test : DTest; objs : array of const) :
Integer;
function replaceIfIn(_start, _end : DIterator; test : DTest; objs : array of const) :
Integer;
```

ReplaceIf replaces items for which the test returns true with objs. You must pass only one item for objs.

Reversing

Reverse

```
procedure reverse(container : DContainer);
procedure reverseIn(_start, _end : DIterator);
```

Reverse reverses the order of items in a sequence. For example, the sequence (1,2,3,4,5) becomes (5,4,3,2,1).

ReverseCopy

```
procedure reverseCopy(con1, con2 : DContainer);
procedure reverseCopyTo(container : DContainer; output : DIterator);
procedure reverseCopyInTo(_start, _end, output : DIterator);
```

ReverseCopy copies a sequence to a new location, reversing it during the copy.

Rotating

Rotate

```
procedure rotate(first, middle, last : DIterator);
```

Rotate performs a right rotation on a sequence. The *first* item will end up at position *middle*, the second at *middle + 1*, and so forth.

RotateCopy

```
function rotateCopy(first, middle, last, output : DIterator) : DIterator;
```

RotateCopy does the same thing as rotate except that the original sequence is unchanged – the rotated result is written to a new location.

Set Operations

Includes

```
function includes(master, subset : DContainer) : Boolean;
function includesWith(master, subset : DContainer; comparator : DComparator) :
Boolean;
function includesIn(startMaster, finishMaster, startSubset, finishSubset : DIterator)
: Boolean;
function includesInWith(startMaster, finishMaster, startSubset, finishSubset :
DIterator; comparator : DComparator) : Boolean;
```

Includes determines if a master set includes an entire sub set. Includes relies on the two containers or ranges being sorted. If set 1 is (1,2,3,4,5) and set 2 is (2,3,4), includes returns true. If set 1 is (1,2,3,4,5) and set 2 is (2,3,10), includes returns false.

SetDifference

```
function setDifference(con1, con2 : DContainer; output : DIterator) : DIterator;
function setDifferenceIn(start1, finish1, start2, finish2, output : DIterator) :
DIterator;
function setDifferenceWith(con1, con2 : DContainer; output : DIterator; comparator :
DComparator) : DIterator;
function setDifferenceInWith(start1, finish1, start2, finish2, output : DIterator;
comparator : DComparator) : DIterator;
```

SetDifference finds the set of items that are in the first range but not in the second range. It sends this new set of items to an output iterator. SetDifference relies on both ranges being sorted. If set 1 is (1,2,3,4,5) and set 2 is (2,3,4), setDifference returns (1,5).

SetIntersection

```
function setIntersection(con1, con2 : DContainer; output : DIterator) : DIterator;
function setIntersectionIn(start1, finish1, start2, finish2, output : DIterator) :
DIterator;
function setIntersectionWith(con1, con2 : DContainer; output : DIterator; comparator
: DComparator) : DIterator;
function setIntersectionInWith(start1, finish1, start2, finish2, output : DIterator;
comparator : DComparator) : DIterator;
```

SetIntersection finds the set of items that are in both containers or ranges. It sends this new list of items to an output iterator. SetIntersection relies on both ranges being sorted. If set 1 is (1,2,3,4,5) and set 2 is (2,3,4,10), setIntersection returns (2,3,4).

SetSymmetricDifference

```
function setSymmetricDifference(con1, con2 : DContainer; output : DIterator) :
DIterator;
function setSymmetricDifferenceIn(start1, finish1, start2, finish2, output :
DIterator) : DIterator;
function setSymmetricDifferenceWith(con1, con2 : DContainer; output : DIterator;
comparator : DComparator) : DIterator;
function setSymmetricDifferenceInWith(start1, finish1, start2, finish2, output :
DIterator; comparator : DComparator) : DIterator;
```

SetSymmetricDifference finds the items that are **not** in both sets. It relies on both ranges being sorted. If set 1 is (1,2,3,4,5) and set 2 is (4,5,6,7,8), setSymmetricDifference returns (1,2,3,6,7,8);

SetUnion

```
function setUnion(con1, con2 : DContainer; output : DIterator) : DIterator;
function setUnionIn(start1, finish1, start2, finish2, output : DIterator) :
DIterator;
function setUnionWith(con1, con2 : DContainer; output : DIterator; comparator :
DComparator) : DIterator;
function setUnionInWith(start1, finish1, start2, finish2, output : DIterator;
comparator : DComparator) : DIterator;
```

SetUnion finds the items that are in both sequences. It relies on both ranges being sorted. Only one copy of each value will be present in the output set. If set 1 is (1,2,3,4,5) and set 2 is (4,5,6,7,8), setUnion will return (1,2,3,4,5,6,7,8).

Shuffling

RandomShuffle

```
procedure randomShuffle(container : DContainer);
```

```
procedure randomShuffleIn(_start, _end : DIterator);
```

RandomShuffle randomly moves around elements in the container, just like shuffling a deck of cards.

Sorting

Sort

```
procedure sort(sequence : DSequence);
procedure sortIn(_start, _end : DIterator);
procedure sortWith(sequence : DSequence; comparator : DComparator);
procedure sortInWith(_start, _end : DIterator; comparator : DComparator);
```

Sort sorts the items in the container or range it is given. This sort is not stable; that is, the ordering the elements have in the container before the sort algorithm is run have nothing to do with the order after the sort is run. Sort is based on a QuickSort.

StableSort

```
procedure stablesort(sequence : DSequence);
procedure stablesortIn(_start, _end : DIterator);
procedure stablesortWith(sequence : DSequence; comparator : DComparator);
procedure stablesortInWith(_start, _end : DIterator; comparator : DComparator);
```

StableSort sorts the items in the container or range, an maintains (without violating sort ordering) the current order of the items in the container. StableSort is based on a MergeSort.

Swapping

IterSwap

```
procedure iterSwap(iter1, iter2 : DIterator);
```

IterSwaps swaps the values two iterators are positioned at.

SwapRanges

```
procedure swapRanges(con1, con2 : DContainer);
procedure swaprangesInTo(start1, end1, start2 : DIterator);
```

SwapRanges swaps the values in two ranges – the values in the first range will move to the second range, and the values in the second range will move to the first.

Transforming

Collect

```
function collect(container : DContainer; unary : DUnary) : DContainer;
function collectIn(_start, _end : DIterator; unary : DUnary) : DContainer;
```

Collect applies the unary function to each object in the container, storing the results in a new container (that is constructed by the function) that is of the same type as the existing one.

TransformBinary

```
procedure transformBinary(con1, con2, output : DContainer; binary : DBinary);
function transformBinaryTo(con1, con2 : DContainer; output : DIterator; binary :
DBinary) : DIterator;
function transformBinaryInTo(start1, finish1, start2, output : DIterator; binary :
DBinary) : DIterator;
```

TransformBinary applies a binary function to pairs of objects from con1 and con2, and stores the result into the output area. con1 and con2 need to have the same number of objects in them.

TransformUnary

```
procedure transformUnary(container, output : DContainer; unary : DUnary);
function transformUnaryTo(container : DContainer; output : DIterator; unary : DUnary)
: DIterator;
function transformunaryInTo(_start, _finish, output : DIterator; unary : DUnary) :
DIterator;
```

TransformUnary applies a unary function to each item in a container or range, and stored the results in an output area.

Utility Functions

SDL provides a number of utility functions to make using the library easier. These mostly revolve around converting atomic types in and out of DObjects, as well as functions to aid in common programming situations.

Atomic Converters

SDL provides a series of functions that can aid you in moving atomic values into and out of the DObject structure, with and without iterators. Each of these functions has many variants, named for the atomic types. XXXX can be any of the following:

Integer, Boolean, Char, Extended, ShortString, Pointer, PChar, Object, Class, WideChar, PWideChar, String, Currency, WideString

```
Function AsXXXX(const obj : DObject) : XXXX;
```

Converts a DObject to the specified type, leaving the original value in place.

```
Function ToXXXX(const obj : Dobject) : XXXX;
```

Converts a DObject to the specified type, clearing the original.

```
Procedure SetXXXX(var obj : Dobject; const value : XXXX) ;
```

Sets the value of an already initialized DObject to a new value. The old value is cleared and freed.

```
Function GetXXXX(const iter : DIterator) : XXXX;
```

Retrieves the DObject at the iterator's position as an XXXX.

```
Procedure PutXXXX(const iter : DIterator, const value : XXXX);
```

Writes the value to the iterator's current position. The old value is cleared and replaced with the new one.

Iterator Helpers

```
function MakePair(const ob1, ob2 : DObject) : DPair;
```

MakePair copies two DObjects into a pair object, which it returns. You need to make sure that you clean up the pair object that is returned.

```
function MakeRange(s,f : DIterator) : DRange;
```

MakeRange converts to iterators into a range. Sometimes it's easier to manipulate ranges directly, inside a DRange structure. Certain algorithms will return ranges as DRanges.

Hashing

```
function hashCode(const obj : DObject) : Integer;
```

Return a hash value for a DObject. The object is hashed according to its type.

```
function JenkinsHashInteger(value : Integer) : Integer;
```

Return a hash value for an integer.

```
function JenkinsHashBuffer(const buffer; length : Integer; initVal : Integer) :
Integer;
```

Return a hash value for a series of bytes. Pass the variable you want to has as buffer. Be careful to note that buffer is an untyped const. If you have a pointer to some variable, and you want to hash the variable, use the ^ notation.

```
function JenkinsHashString(const s : String) : Integer;
```

Return the hash value for a string.

```
function JenkinsHashSingle(s : Single) : Integer;
```

Return the hash value for a **single** value.

```
function JenkinsHashDouble(d : Double) : Integer;
```

Return the hash value for a **double** value.

DObject Helpers

SDL provides a number of helper functions for getting objects into and out of DObjects. You need to pay some attention to the lifetimes and initialization states of your DObjects. In particular, you need to make sure that you *never store a string value to an uninitialized DObject*. Most of the time, if you store a value to an uninitialized DObject, it won't make much difference. Delphi does reference-counted strings, though, so storing a string value to a random piece of memory can cause an access violation.

There are two easy ways to get around this. The first is not to use the SetString function, unless you're *sure* the DObject in question has already been initialized. The second is to use the Make function or the CopyDObject function, which ensure that the destination is initialized before storing a value.

You need to be particularly aware of this when you are creating callbacks that return DObjects. The result variable (which is often a DObject), is *not* initialized when your procedure gets it. You need to make sure you *clear* result, or assign it with the Make function or the CopyDObject function.

```
Function Make(value : array of const) : DObject;
```

Creates a new DObject, based on the value you supply. You are responsible for cleaning up the storage of this DObject, if necessary. This function is frequently used to return the results of callback functions.

```
procedure InitDObject(var obj : DObject);
```

Empties a DObject, ensuring that it is ready to receive whatever you want to put in it. The previous contents of the object are *not* freed or cleared. If you want to do that, use ClearDObject, or one of the SetObjectXXX family.

```
procedure CopyDObject(const source : DObject; var dest : DObject);
```

Copies a DObject from source to dest. The destination is initialized before writing the new value. Any object that was in destination is lost, and is not cleared.

```
procedure MoveDObject(var source, dest : DObject);
```

Moves a DObject from the source to the dest. The destination is initialized before writing the new value. Any object that was in the destination is lost. After a copy, the source is cleared.

```
procedure ClearDObject(var obj : DObject);
```

Frees any storage and clears the object, resetting it to an initialized state. The DObject is then ready to receive another value.

```
procedure SetDObject(var obj : DObject; value : array of const);
```

Sets a DObject to any atomic value. Clears the DObject first, releasing any storage currently being used by the DObject. Do not call any function in the SetXXX family unless you are sure that the target DObject has been initialized.

```
procedure Swap(var obj1, obj2 : DObject);
```

Swaps the values of any two DObjects.

Morphing Closures

SDL can use BOTH closures (procedures of object) and regular functions for those places in which it needs to call your code. It does this by taking advantage of the way that Delphi's object model works. Let's look at an example:

```
DComparator = function(const obj1, obj2 : DObject) : Integer of object;
```

That's the official definition of DComparator. SDL also provides the following:

```
DComparatorProc = function(ptr : Pointer; const obj1, obj2 : DObject) : Integer;
```

These two definitions amount to the same call in Delphi. On the closure (the first one), Delphi passes an _invisible parameter_, self, as the first parameter to the call. Self is always a pointer. In the second one, we are making the pointer an explicit part of the call.

SDL also provides these:

```
function MakeComparator(proc : DComparatorProc) : DComparator;
begin
TMethod(result).data := nil;
TMethod(result).code := @proc;
end;
function MakeComparatorEx(proc : DComparatorProc; ptr : Pointer) : DComparator;
begin
TMethod(result).data := ptr;
TMethod(result).code := @proc;
end;
```

We can then do something like this:

```
function MyComparator(ptr : Pointer; const obj1, obj2 : DObject) : Integer;
begin
...
end;
x := DArray.CreateWith(MakeComparator(MyComparator));
or
x := DArray.CreateWith(MakeComparatorEx(MyComparator, a_pointer_i_want_to_pass));
```

Now -- why do we want all this? Simple -- most Delphi code is done in methods on forms or on objects. SDL needs to have a way to make callbacks onto methods on those objects, and that led to the requirement that closures be part of the definitions. But, with the techniques outlined above, we can also use regular functions as callbacks, which are very useful for putting small bits of code right near where they're used.

```
function MakeComparator(proc : DComparatorProc) : DComparator;
function MakeEquals(proc : DEqualsProc) : DEquals;
function MakeTest(proc : DTestProc) : DTest;
function MakeApply(proc : DApplyProc) : DApply;
function MakeUnary(proc : DUnaryProc) : DUnary;
function MakeBinary(proc : DBinaryProc) : DBinary;
function MakeHash(proc : DHashProc) : DHash;
function MakeGenerator(proc : DGeneratorProc) : DGenerator;
```

These are the definitions for the functions that can create closures out of regular procedures.

```
function MakeComparatorEx(proc : DComparatorProc; ptr : Pointer) : DComparator;
function MakeEqualsEx(proc : DEqualsProc; ptr : Pointer) : DEquals;
function MakeTestEx(proc : DTestProc; ptr : Pointer) : DTest;
function MakeApplyEx(proc : DApplyProc; ptr : Pointer) : DApply;
function MakeUnaryEx(proc : DUnaryProc; ptr : Pointer) : DUnary;
function MakeBinaryEx(proc : DBinaryProc; ptr : Pointer) : DBinary;
function MakeHashEx(proc : DHashProc; ptr : Pointer) : DHash;
function MakeGeneratorEx(proc : DGeneratorProc; ptr : Pointer) : DGenerator;
```

Sometimes it’s useful to be able to pass a pointer to the procedure you’re making a closure for. The Ex versions of these functions allow you to do just that. The pointer you put in will be passed to your procedure as its first parameter.

Printing

SDL has some built-in support for printing the contents of containers. This is often useful during the debugging phase of developing your application. SDL knows how to print the basic types, but if you want it to print your own objects, you’ll need to register a printing routine. The printing routine has this signature:

```
DPrinterProc = function (obj : TObject) : String;
```

After you create a routine with that signature, you’ll need to call the following routine to register it with SDL:

```
procedure RegisterSDLPrinter(cls : TClass; prt : DPrinterProc);
```
Pass the class object in as the first parameter, like this:

```
Function MyPrinter(obj : TObject) : String;
Begin
```

```
With obj as TMyClass do
  …
end;
RegisterSDLPrinter(TMyClass, MyPrinter);
```

SDL provides a helper function to convert a DObject into a printable string. This function will call any registered printing functions for objects that it encounters.

```
function PrintString(const obj : DObject) : String;
```

Printing is often done in conjunction with the forEach routine, which can apply a printing function to each item in a container. SDL provides the ApplyPrint routine, which can be passed directly to forEach for a container or range, and invokes PrintString to get the strings it needs to write to the console.

```
procedure ApplyPrint(ptr : Pointer; const obj : DObject);
```

The ApplyPrintLN variant puts a linefeed after it prints each item. This is nice when you're print objects.

```
procedure ApplyPrintLN(ptr : Pointer; const obj : DObject);
```

Debugging Support

SDL contains numerous assertions throughout its code. In the binary release of the library, these assertions are turned off. If you have the source version, you can recompile the library and turn the assertions on. They will catch many of the common problems that you will encounter while using SDL.

Any time that SDL throws an exception, you can expect that something has gone quite wrong. SDL does not throw exceptions during the course of any normal activity; for this reason, you never need to turn off "break on exception" while working with SDL. Most SDL exceptions will contain a message indicating what went wrong.

Persistence with SuperStream

SuperStream is SDL's companion library. It provides simple, powerful object streaming capabilities. The object streams support atomic types, objects, inheritance and permits the storage and loading of multiple versions of objects. Object graphs (arbitrarily connected sets of objects) are also supported, as an option. SuperStream's primary advantages over other streaming systems are:

- Ease of use
- Nested object support
- Source not required to stream an object's data
- Intelligent, atomic-type aware transfer mechanism
- Object versioning
- SuperStream can effectively save and load SDL containers, as well.

To use SuperStream, you need only provide a simple *Transfer Function* for each of your classes.

Basic Concepts

Stream

Streams are Delphi's official way of handling most I/O. Delphi provides a number of basic stream classes, like TMemoryStream, TFileStream, and so forth. They all have as their base class TStream. SuperStream creates a subclass of TStream called TStreamAdapter, which is designed to *wrap one stream with another*. This allows us to add additional behavior onto an existing stream. This layering is a very powerful abstraction, and it permits SuperStream to act as efficiently and flexibly as it does.

Object

The root of most of your data in Delphi will be the object. SuperStream can save and load Delphi's atomic types, and can also save and load objects. One of the advantages of SuperStream is that it does not require you to derive the classes you want to save and load from a common base class. It also doesn't require that you have the source code to these classes. You only need to provide a Transfer Function, which is independent of the class.

Atomic Types

Atomic types are Delphi's fundamental types, such as String, Integer, Extended, ShortString, and so on. SuperStream knows how to read and write most of these types automatically, so you don't need to do very much work. Certain atomic types, such as Variants, cannot be streamed in and out. Your transfer function may have to do some extra work to save and load these types.

Transfer Function

A transfer function (also known as an IO procedure) is a simple function that tells SuperStream how to save and load your objects. The transfer function has been designed to be as simple and fast to implement as possible. Let's look at one now, so you can see how simple it can be. What follows is a type definition and a transfer function for that type.

```
TTest = class
public
  s,t : String;
end;
procedure TestIO(obj : TObject; stream : TObjStream; direction : TObjIODirection;
version : Integer; var callSuper : Boolean);
begin
   with obj as TTest do stream.TransferItems([s,t], [@s, @t], direction, version);
end;
```

This transfer function (TestIO), can both read and write any TTest object. The more advanced capabilities of SuperStream aren't used in this simple example, which is intended to show the brevity that is possible.

Note that separate read and write routines are not necessary. You should also note that all the fields in the object were read and written with a single, simple call. This is the power of SuperStream! You can easily create transfer functions for your classes.

Object Versioning

As an application changes and improves, it often finds itself adding fields to its objects, or altering them in some way. SuperStream attaches a version number to each object that it writes to a stream. When the object is read back in, the version number is passed to the transfer function, which can read the old version and make appropriate changes to make the object compatible with the newer one.

This "automatic upgrading" of objects is very convenient when maintaining an application. Usually, an application will read old versions of objects, and automatically upgrade them to the latest version when they are stored.

Buffered Stream

Delphi's streaming functions for files are useful, but tend to be rather slow when many small read and write calls are made. SuperStream makes many, many of these kinds of calls. To get around this problem, SuperStream provides buffering stream adapters. These stream adapters wrap themselves around another stream (like TFileStream), and add a buffering capability to accelerate operations on the stream. On large reads or writes, with many thousands of objects, order-of-magnitude or higher speedups are gained.

Of course, on TMemoryStreams buffering isn't necessary.

Nine Easy SuperStream Lessons

Just like SDL, we'll introduce the SuperStream library with simple lessons. These will provide simple narratives that will describe the problem to be solved, and demonstrate how

it is solved with SuperStream. After the lessons, you'll find more detailed reference information on the library and the classes it contains.

Lesson 1 – Saving and Loading One Object

Here we'll tackle the simplest case: We have an object that we want to save into a file, and then read it back. We'll use an object called TTest for this sample.

```
TTest = class
public
   s,t : String;
   yipe : Integer;

   constructor Create;
end;

constructor TTest.Create;
begin
  s := 'zonk';
  t := RandomString;
  yipe := Integer(self);
end;

procedure TestIO(obj : TObject; stream : TObjStream; direction : TObjIODirection;
version : Integer; var callSuper : Boolean);
begin
   with obj as TTest do stream.TransferItems([s, t, yipe], [@s, @t, @yipe],
direction, version);
end;

procedure SimpleExample;
var test : TTest;
begin
   TObjStream.RegisterClass(TTest, TestIO, 1);
   Test := TTest.Create;
   TObjStream.WriteObjectToFile('simple.od', [], test);
   Test.free;
   Test := TObjStream.ReadObjectInFile('simple.od', []) as TTest;
   // We're done!
   Test.free;
end;
```

Let's take apart this sample code, so we can see how all this works. The first thing we did is define our class, TTest. We made a simple constructor on TTest to put some random information into the object. Then we defined TestIO – the transfer function for our TTest class. Transfer functions are what you need to write to make SuperStream work for you, so let's look at the function in more detail.

A transfer function (TObjIO) has the following signature:

```
TObjIO = procedure(obj : TObject; stream : TObjStream; direction : TObjIODirection;
version : Integer; var callSuper : Boolean);
```

Your transfer function will always receive a pointer (*obj*) to the object that is being read or written. If an object is being read from the stream, it will already have been constructed by

the time your transfer function is called. You only need to be concerned about making sure the fields in the object are properly written or read.

Stream is the object stream the operation is being performed on. Stream is passed to you so that you can call its methods to help you do the IO. *Direction* indicates whether the object is being read (iodirRead) or written (iodirWrite). For most transfer functions, you don't need to be concerned about whether you are reading or writing. For some specialized functions (such as transferring your own container classes), you may need to know whether a read or write is in progress. *Version* indicates the version of the object that needs to be read or written. When an object is read off a stream, the version number is passed to the transfer routine so that it can elect to read in an old version if necessary, and upgrade the object to the latest version. *CallSuper* is an advanced variable – you only need to be concerned with this if you want to prevent SuperStream from calling a superclass' transfer function. For normal usage, you want to permit SuperStream to take advantage of your class hierarchy.

On to the example! The first thing we do in SimpleExample is register our transfer function, with TObjStream.RegisterClass. This is a necessary step for any IO. It's also a good a idea to call TObjStream.RegisterDefaultClasses. SuperStream knows how to transfer some of the simple VCL classes, and RegisterDefaultClasses tells SuperStream to use these default transfer functions.

RegisterClass takes three parameters. The first is the name of the class you want to register the transfer function for. The second is the transfer function. The third is the *tip version* of the object. When reading objects, the version number comes from the object's definition in the stream. When writing objects, SuperStream will always write the tip version, unless you request otherwise.

Each time you change your object's structure, you should modify your transfer function to read and write the new version, and increment the tip version number. We'll explain this mechanism in more detail, later.

After registering our transfer function, we create a simple object. Then, we write the object to a file. TObjStream provides two helper functions for the very common scenario of writing an object to a file: WriteObjectToFile and ReadObjectInFile. The helper functions do all the work of opening the file stream, wrapping it with a buffered stream, wrapping the buffered stream with an object stream, transferring the object, and then shutting down correctly.

After writing the object, we free our test object, then read the object back in with ReadObjectInFile. Since ReadObjectInFile returns a TObject, we need to cast the object to the correct type. And that's how easy it is to use SuperStream!

WriteObjectToFile and ReadObjectInFile are both *class methods* on TObjStream. That means that you don't need to create a TObjStream object to use them.

As a final point in this lesson, please note that *SuperStream does not call constructors during object reading.* If it's necessary to call an object's constructor to perform some kind of initialization, check to see that you're reading (direction = iodirRead), and then call the constructor on the object directly. You can do this by calling obj.Create, or whatever your

constructor's name is. Calling obj.Create directly bypasses the allocation of a new object and just invokes the construction code.

In our next lesson, we'll examine writing more than one object into a stream.

Lesson 2 – Storing Different Objects

In this lesson we're going to store more than one object into a stream. We're also going to store objects of different classes, and examine how SuperStream deals with that situation. We're also going to take our first look at SuperStream's inheritance mechanism.

Let's assume that we have the same simple TTest type as we defined in the first example. We'll add a second type for this lesson.

```
TExtra = class(TTest)
public
  d : Integer;
end;
```

Note that this type is a *subclass* of TTest. It adds a single field, **d**, to TTest's definition. Here's the IO procedure for TExtra:

```
procedure ExtraIO(obj : TObject; stream : TObjStream; direction : TObjIODirection;
version : Integer; var callSuper : Boolean);
begin
   with obj as TExtra do
   stream.TransferItems([d], [@d], direction, version);
end;
```

Notice that this IO procedure only deals with the field that's been added, **d**. It relies on the superclass' IO procedure to take care of the other fields.

Now let's create a routine that puts a TTest, a TExtra, and another TText object into a memory stream, then reads them back.

```
Procedure ThreeObjects;
Var t1, t2 : TTest;
    E1 : TExtra;
    Ms : TMemoryStream;
    Os : TObjStream;
Begin
  TObjStream.RegisterClass(TTest, TestIO, 1);
  TObjStream.RegisterClass(TExtra, ExtraIO, 1);
  T1 := TTest.Create;
  T2 := TTest.Create;
  E1 := TExtra.Create;
  Ms := TMemoryStream.Create;
  Os := TObjStream.Create(ms, false, []);
  Os.WriteObject(t1);
  Os.WriteObject(e1);
  Os.WriteObject(t2);
  FreeAll([t1, t2, e1]);
  Os.free;
  Ms.position := 0;
  Os := TObjStream.Create(ms, true, []);
```

```
  T1 := os.readObject as TTest;
  E1 := os.readObject as TExtra;
  T2 := os.readObject as TTest;
  Os.free;
End;
```

The first thing we do is register our two classes, followed by the creation of our test objects. We then create the memory stream we want to write into, and write our objects to the stream. Then we free the objects, and reset the memory stream's position to zero.

Note that when we opened the object stream for writing, the second parameter on the constructor was *false*. This parameter tells the object stream whether it *owns* the stream it is wrapping. If the object stream owns the other stream, it will free the other stream when the object stream is freed.

To read our objects back in, we simple create our object stream, then call the stream's ReadObject routine, casting the results to the correct type. Note that we created the object stream with a *true* value for the *owned* parameter. When we free this object stream, it will automatically free the underlying memory stream.

When the TExtra object is written and read, SuperStream first calls its registered IO procedure, ExtraIO. It then walks up the inheritance hierarchy, calling each IO procedure it finds registered. The superclass of TExtra is TTest, so TTest's IO procedure is called next. For this reason, make sure you don't read or write a superclass' fields in an IO procedure, unless you set the CallSuperIO parameter to false, which will prevent walking up the inheritance tree any further.

Lesson 3 – Writing Embedded Objects

One of the best features of SuperStream is that writing embedded objects is no different from writing other atomic types! No special call is needed, and you don't need to treat the object fields differently from other fields. For that reason, this lesson is particularly short. We're going to define a new type that has embedded pointers to other objects in it, and perform some basic IO with it.

```
Type
TEmbed = class
  Int1, int2 : Integer;
  T : TTest;

  Constructor Create;
End;

Constructor TEmbed.Create;
Begin
  Int1 := Random(1000);
  Int2 := Random(1000);
  T := TTest.Create;
End;

procedure EmbedIO(obj : TObject; stream : TObjStream; direction : TObjIODirection;
version : Integer; var callSuper : Boolean);
begin
```

```
  with obj as TEmbed do
      stream.TransferItems(
          [int1, int2, t],
          [@int1, @int2, @t],
          direction, version);
end;

procedure EmbedExample;
var e : TEmbed;
begin
  TObjStream.RegisterClass(TTest, TestIO, 1);
   TObjStream.RegisterClass(TEmbed, EmbedIO, 1);
   E := TEmbed.Create;
   TObjStream.WriteObjectToFile('test.out', [], e);
   e.free;
   e := TObjStream.ReadObjectInfile('test.out', []) as TEmbed;
end;
```

And that's all there is to it! SuperStream automatically detects that the t field is an object, and invokes its IO procedure automatically. Notice that the embedded t object gets full object versioning and all other facilities, as well.

Lesson 4 – Inheritance and SuperStream

SuperStream automatically handles most inheritance issues, because it knows how to call the IO procedures that are registered for any superclasses of an object being read or written. In this example, we'll demonstrate a simple inheritance situation, and a slightly more complex one, in which we don't want the superclass' IO procedure to be called.

```
Type
  TBase = class
    I1, i2 : Integer;
   End;

   TDerived = class(TBase)
     S : String;
   End;

   TAnother = class(TDerived)
    Toast : String;
   End;

procedure BaseIO(obj : TObject; stream : TObjStream; direction : TObjIODirection;
version : Integer; var callSuper : Boolean);
begin
  with obj as TBase do
   stream.transferItems([i1, i2], [@i1, @i2], direction, version);
end;

procedure DerivedIO(obj : TObject; stream : TObjStream; direction : TObjIODirection;
version : Integer; var callSuper : Boolean);
begin
  with obj as TDerived do
    stream.transferItems([s], [@s], direction, version);
end;
```

```
procedure AnotherIO(obj : TObject; stream : TObjStream; direction : TObjIODirection;
version : Integer; var callSuper : Boolean);
begin
   with obj as TAnother do
     stream.transferItems([i1, toast], [@i1, @toast], direction, version);
   callSuper := false;
end;

procedure InheritanceExample;
var b : TBase;
    d : TDerived;
    a : TAnother;
begin
   TObjStream.RegisterClass(TBase, BaseIO, 1);
   TObjStream.RegisterClass(TDerived, DerivedIO, 1);
   TObjStream.RegisterClass(TAnother, AnotherIO, 1);

   b := TBase.Create;
   d := TDerived.Create;
   a := TAnother.Create;

   b.i1 := 100;
   b.i2 := 101;
   TObjStream.WriteObjectToFile('base.od', [], b);

   d.i1 := 200;
   d.i2 := 201;
   d.s := 'Hello';
   TObjStream.WriteObjectToFile('derived.od', [], d);

   a.i1 := 300;
   a.i2 := 301;
   a.s := 'yarf';
   a.toast := 'toast';
   TObjStream.WriteObjectToFile('another.od', [], a);

   FreeAll([b,d,a]);

   b := TObjStream.ReadObjectInFile('base.od', []) as TBase;
   D := TObjStream.ReadObjectInFile('derived.od', []) as TDerived;
   A := TObjStream.ReadObjectInFile('another.od', []) as TAnother;

   Writeln('base: ', b.i1, ' ', b.i2);
   Writeln('derived: ', d.i1, ' ', d.i2, ' ', d.s);
   Writeln('another: ', a.i1, ' ', a.i2, ' ', a.s, ' ', a.toast);

   FreeAll([b,d,a]);
end;
```

So what do we expect to be printed out by this example? We expect this:

```
Base: 100 101
Derived: 200 201 Hello
Another: 300 0 toast
```

TAnother's IO procedure is preventing the calling of the base class IO procedures, so some of the fields are not written. You can use this technique when you want your IO procedure to take charge of all IO for an object, preventing the subclass from doing anything.

Lesson 5 – Storing SDL Containers

Activating SDL's integration with SuperStream is very simple. Just add the SDLIO unit into your project, and all SDL container classes will automatically be registered for streaming. You'll still need to code IO procedures for your own classes, but SDL will take care of itself.

SDL's container class design and SuperStream are highly complimentary. To perform IO on all 13 container classes in SDL, only two IO procedures needed to be written. One IO procedure handles containers that are single data element oriented (like arrays and lists), and the other handles containers that are pair oriented (like maps and hash maps). SuperStream's inheritance mechanism and SDL's virtual constructors ensure that the correct results are reached. Here's an example of an SDL container saving and loading itself automatically:

```
Uses SDLIO;  // causes automatic registration of SDL-SuperStream integration
Procedure SDLIOExample;
Var c : DContainer;
    I : Integer;
Begin
   C := DArray.Create;
   For I := 1 to 20 do
      Begin
       Case I mod 3 of
        0 : c.add([I]);
         1 : c.add([IntToStr(I)]);
         2 : c.add(TTest.Create);
      End;
  TObjStream.WriteObjectToFile('container.od', [], c);
  ObjFree(c);
  c.free;
  c := TObjStream.ReadObjectInFile('container.od', []) as DContainer;
end;
```

This example also demonstrates how SuperStream and SDL can deftly handle the persistence of a container class that contains different types (some of which are atomic, and some of which are objects). SuperStream automatically checks all objects in the SDL container and performs the correct kind of IO on them.

Lesson 6 – Storing Special Types (TDateTime, Single, Double)

Inprise's array of const is the core trick at the base of SDL and SuperStream. We can make this mechanism do a great deal of work for us, but unfortunately it doesn't do everything. The place it falls down a bit is in dealing with different floating point types. The array of const mechanism automatically casts each floating point value into an Extended value. Because it does this, we cannot distinguish between Single, Double, TDateTime (which is based on Double), and Extended. To get around this limitation, SuperStream provides some extra type codes (called the ssvt constants), and a more powerful version of TransferItems,

TransferItemsEx. The two transferItems calls are identical, except for an addition open array parameter, which specifies the *type codes* for the items being written.

To understand the type codes, you need to understand what Delphi does when it creates an array of const. Each item passed in the array gets put in a TVarRec structure, which is defined like this:

```
TVarRec = record
    case Byte of
      vtInteger:    (VInteger: Integer; VType: Byte);
      vtBoolean:    (VBoolean: Boolean);
      vtChar:       (VChar: Char);
      vtExtended:   (VExtended: PExtended);
      vtString:     (VString: PShortString);
      vtPointer:    (VPointer: Pointer);
      vtPChar:      (VPChar: PChar);
      vtObject:     (VObject: TObject);
      vtClass:      (VClass: TClass);
      vtWideChar:   (VWideChar: WideChar);
      vtPWideChar:  (VPWideChar: PWideChar);
      vtAnsiString: (VAnsiString: Pointer);
      vtCurrency:   (VCurrency: PCurrency);
      vtVariant:    (VVariant: PVariant);
      vtInterface:  (VInterface: Pointer);
      vtWideString: (VWideString: Pointer);
end;
```

By setting the VType field and the other fields, the TVarRec allows just about any atomic type to be represented. There's a table of type codes, called vt constants. They are as follows (and are defined in the system unit):

```
vtInteger    = 0;
vtBoolean    = 1;
vtChar       = 2;
vtExtended   = 3;
vtString     = 4;
vtPointer    = 5;
vtPChar      = 6;
vtObject     = 7;
vtClass      = 8;
vtWideChar   = 9;
vtPWideChar  = 10;
vtAnsiString = 11;
vtCurrency   = 12;
vtVariant    = 13;
vtInterface  = 14;
vtWideString = 15;
```

SuperStreams adds a couple of new type codes:

```
ssvtSingle = -2;
ssvtDouble = -3;
ssvtDateTime = ssvtDouble;
```

Normally, when you call TransferItems, you don't need to specify type codes, because Delphi supplies them for you when you create an array of const. For the special types (single, double, TDateTime), you need to tell SuperStream the type of the variable.

The third parameter of the TransferItemsEx call is the special one. It specifies the vt type codes for each of the fields you are writing. SuperStream provides a shortcut here – frequently you aren't writing that many of these special fields. You don't need to specify the type code for all of the fields you are writing. To take advantage of this, put all your special fields **first**, supplying type codes in the third parameter for them. SuperStream will automatically assign the remaining type codes. Our code example for this lesson will demonstrate all of this.

```
Type
TSpecial = class
  Int1, int2 : Integer;
  St : String;
  When : TDateTime;
  R : Single;
End;

procedure SpecialIO(obj : TObject; stream : TObjStream; direction : TObjIODirection;
version : Integer; var callSuper : Boolean);
begin
   with obj as TSpecial do
       stream.TransferItemsEx(
           [when, r, int1, int2, st],
           [@when, @r, @int1, @int2, @st],
           [ssvtDateTime, ssvtSingle],
           direction, version);
end;

procedure SpecialExample;
var s : TSpecial;
begin
   TObjStream.RegisterClass(TSpecial, SpecialIO, 1);
   s := TSpecial.Create;
   with s do
     begin
       s.int1 := Random(1000);
       s.int2 := Random(1000);
       s.st := RandomString;
       s.when := Now;
        s.r := Random(1000) / 1000;
     end;
   TObjStream.WriteObjectToFile('test.out', [], s);
   S.free;
   S := TObjStream.ReadObjectInFile('test.out', []) as TSpecial;
    s.free;
end;
```

Note that we only specified the special type codes for the first two variables, where it was necessary to do so.

Lesson 7 – Storing Raw Data

SuperStream provides a number of facilities to help with the transfer of raw data in and out of streams. Storing your application data sometimes requires this handling of large blocks of data for objects like bitmaps, or if you have a class with an embedded array, or things of that type.

We're going to demonstrate how to do IO on arbitrary blocks of memory, and how to transfer arrays of atomic types. What we'll show here is a sample type that has both an array of strings in it, and a bunch of raw data that represents a bitmap. We want to read and write this object.

```
Type
TRaw = class
  FNames : Integer;
  FName : array[1..25] of String;
  FAddresses : array[1..25] of String;
  FEmployees : array[1..25] of TEmployee;
  FBitmapSize : Integer;
  FBitmapData : Pointer;
End;

procedure SpecialIO(obj : TObject; stream : TObjStream; direction : TObjIODirection;
version : Integer; var callSuper : Boolean);
begin
   with obj as TRaw do
     begin
       stream.TransferItems([FNames, FBitmapSize] [@Fnames, @FBitmapSize], direction,
version);
       stream.TransferArrays(
        [FName[1], FAddresses[1], FEmployees[1]],
         [@FName[1], @FAddresses[1], @FEmployees[1]],
         [25, 25, 25],
         direction);
       if direction = iodirRead then
          GetMem(FBitmapData, FbitmapSize);
       stream.TransferBlocks([FBitmapData], [FBitmapSize], direction);
     end;
end;

procedure RawExample;
var r : TRaw;
begin
   r := TRaw.Create;
   TObjStream.RegisterClass(TRaw, RawIO, 1);
   TObjStream.WriteObjectToFile('raw.od', [], r);
   r.free;
   r := TObjStream.ReadObjectInFile('raw.od', []) as TRaw;
   r.free;
end;
```

This example shows the general technique for reading and writing arbitrary arrays of atomic values, and storing binary blocks of data. Everything here should be familiar except what's new in the IO procedure, so let's concentrate on that.

The first thing this IO procedure does it transfer the sizes of the other items it's going to write. These are atomic values and are very simple to move, so we do them first. We also do them first because we may need to get at the information in them during a read operation.

Next we invoke the TransferArrays function. TransferArrays needs four parameters: the **first** item in each array (which is used to get type information), the **address** of the first item in the array (which is used to figure out where everything is), the **number** of items in the array, and the direction flag. That's all that's needed – SuperStream takes care of figuring out the rest, and handles the transfer of all atomic values (including arrays of objects) automatically.

Following that is a transfer of a binary block of data. This is accomplished with the TransferBlocks function. TransferBlocks can actually write multiple blocks at the same times (just like TransferArrays can write multiple arrays simultaneously). In our example we're only writing one block.

The only wrinkle in this example is that during a **read**, we need to allocate the memory for our block. This is accomplished with the GetMem call – and note that we already know the size of the block we're reading, because it was part of the atomic value read we did at the beginning of the IO procedure.

TransferBlocks takes three parameters: The address of the block, the size of the block, and a direction flag. SuperStream takes care of the rest.

Lesson 8 – Storing Complex Object Graphs

You may have a *complex object graph* – which is a bunch of objects that have pointers that refer to each other. SuperStream can take care of this for you automatically, by assembling an object graph tracking system. To enable this mechanism (which slows down SuperStream very slightly), pass the [osoGraph] option to the constructor of the TObjStream. Or, if you're using the WriteObjectToFile or ReadObjectInFile calls, pass [osoGraph] as the options.

This short example will demonstrate this:

```
Type
TOne = class;
TTwo = class;
TOne = class
  FHello : String;
  Inside : TTwo;
End;
TTwo = class
  Inside : TOne;
  Ouch : String;
End;

procedure OneIO(obj : TObject; stream : TObjStream; direction : TObjIODirection;
version : Integer; var callSuper : Boolean);
begin
   with obj as TOne do
     stream.TransferItems([FHello, Inside], [@Fhello, @Inside], direction, version);
end;
```

```
procedure TwoIO(obj : TObject; stream : TObjStream; direction : TObjIODirection;
version : Integer; var callSuper : Boolean);
begin
   with obj as TTwo do
     stream.TransferItems([Ouch, Inside], [@Ouch, @Inside], direction, version);
end;

procedure GraphExample;
var o : TOne;
    t : TTwo;
begin
   TObjStream.RegisterClass(TOne, OneIO, 1);
   TObjStream.Registerclass(TTwo, TwoIO, 1);
   O := TOne.Create;
   T := TTwo.Create;
   o.FHello := 'hello';
   o.inside := t;
   t.ouch := 'ouch';
   t.inside := o;

   TObjStream.WriteObjectToFile('test.od', [osoGraph], o);
   FreeAll([o,t]);

   O := TObjStream.ReadObjectInFile('test.od', [osoGraph]) as TOne;

   Writeln('Proof: ', o.inside.inside.Fhello);
end;
```

By passing the osoGraph option, SuperStream keeps track of all objects it reads an writes. It can then properly restore multiple references to the same object. In this example, t's pointer to o is automatically set up, even though o has already been read.

Lesson 9 – Reading and Writing Different Versions of Objects

Objects change as an application is maintained. We're going to look at how SuperStream deals with different versions of objects in this example. What we'll do is define an object, show its IO procedure, then change the definition of the object and show the new IO procedure for it.

```
Type
TAppObject = class
  Name, address : String;
  Salary : Integer;
End;

procedure AppIO(obj : TObject; stream : TObjStream; direction : TObjIODirection;
version : Integer; var callSuper : Boolean);
begin
   with obj as TAppObject do
     stream.transferItems([name, address, salary], [@name, @address, @salary],
direction, version);
end;

procedure InitExample;
var a : TAppObject;
```

```
begin
   TObjStream.RegisterClass(TAppObject, AppIO, 1);
    A := TAppObject.Create;
    TObjStream.WriteObjectToFile('zot', [], a);
    a.free;
    a := TObjStream.ReadObjectInFile('zot', []) as TAppObject;
end;
```

This is a very simple IO procedure, and very straightforward. Now we'll make two changes to the existing object: We'll add a new field and we'll delete an old one. Here's what the new code looks like:

```
Const
HighSalaryValue = nnnn;
Type
TAppObject = class
    Name, Address : STring;
   SalaryHigh : Boolean;
End;

procedure AppIO(obj : TObject; stream : TObjStream; direction : TObjIODirection;
version : Integer; var callSuper : Boolean);
var oldSalary : Integer;
begin
  case version of
   1: with obj as TAppObject do
     begin
       stream.transferItems([name, address, oldSalary], [@name, @address,
@oldSalary], direction, version);
       SalaryHigh := oldSalary > HighSalaryValue;
     end;
   2:
   with obj as TAppObject do
     stream.transferItems([name, address, salaryHigh], [@name, @address,
@salaryHigh], direction, version);
end;

procedure InitExample;
var a : TAppObject;
begin
   TObjStream.RegisterClass(TAppObject, AppIO, 2);
    A := TAppObject.Create;
    TObjStream.WriteObjectToFile('zot', [], a);
    a.free;
    a := TObjStream.ReadObjectInFile('zot', []) as TAppObject;
end;
```

We've modified the IO procedure to have a case statement, switched on the version being passed in. If we're reading or writing the tip version (which has now been set to 2 – notice the change in the RegisterClass call), then we just perform convention IO. If we're reading an old object (which is pretty much the only way it happens, because writes of old objects don't happen that much), we need to do a little extra processing. We need to make sure we correctly read in any deleted members from the old version of the object. Then, we need to make sure that any new members are filled in correctly. We do this in this case by testing the salary value that's read in, and setting the field as appropriate.

This technique of using temporary variables to hold old, deleted values works well. In practice, you'll rarely be deleting variables. Addition of new fields is much more common. Just make sure that your IO procedure can correctly initialize the new values.

A very important point to note is that *constructor functions are not called by SuperStream*. SuperStream relies on the IO procedure to correctly create the fields. If it's really important that the constructor be called, check to see if you're reading an object (direction = iodirRead) and then invoke the constructor yourself. SDLIO does this to ensure that the container classes are correctly built. *Make sure that you only call the constructor if the io direction is iodirRead*. The author of SDL and SuperStream got bitten badly by this one ☺.

SuperStream Classes

This section contains a brief discussion of the classes that are included in the SuperStream unit, and how you would use them. See the Lesson material for more detailed examples on usage, and common situations. The online documentation contains a detailed HTML reference for the SuperStream classes; please refer to it for complete information. This section discusses the major points you should be aware of.

TStreamAdapter

We discuss TStreamAdapter first because it is the root of the SuperStream class hierarchy. A stream adapter is a TStream-compatible object that "wraps" itself around another stream, usually to provide additional functionality. SuperStream's TObjStream class is a stream adapter. Using the adapter technique permits TObjStream to read and write objects from any other type of TStream, including TFileStream and TMemoryStream, which are shipped with Delphi. Using an adapter stream also permits TObjStream to operate over other types of streams that may not be included with Delphi. Examples of such streams include compressing streams and buffering stream adapters, or streams that write their contents over network connections.

The TStreamAdapter provides a constructor that takes another stream as an argument; this is the stream that is being "wrapped". It then delegates the TStream functions onto the wrapped stream. This provides a basis for creating new adapter streams.

TObjStream

TObjStream is the primary focus of the SuperStream package. It provides comprehensive support for reading and writing objects to and from streams. It also provides many convenience functions for handling large binary objects, handling arrays of atomic types, and dealing with complex graphs of objects. The Lessons section contains detailed examples of the proper usage of TObjStream. For comprehensive reference information, please see the online HTML reference.

TObjStream is descended from TStreamAdapter, so that it can be wrapped around any kind of stream. If you are reading and writing from files, Soletta highly recommends that you first wrap the TFileStream with one of the buffering stream adapters, then wrap the object stream around the buffered stream. You may achieve an order of magnitude performance improvement, or more, by doing this.

The binary data created by TObjStream compresses well, so you may wish to add a compressing stream to your application's design.

TBufferedInputStream

TBufferedInputStream is designed to provide buffered read access to another stream. Seeking and writing are not supported – the sole purpose of this class is to provide rapid, sequential access to the data contained in another stream.

TBufferedOutputStream

TBufferedOutputStream is designed to provided buffered write access to another stream. Reading and seeking are not supported, and will cause exceptions to be thrown. This class is intended to provide output buffering of a sequentially written stream. The output of TObjStream has this characteristic.

TObjList

TObjList is a simple list class provided by SuperStream to contain a list of objects. It is a subclass of Delphi's TList class, with an addition property or two. You may find it useful in your "quick and dirty" apps, if you don't want to bring in the full power of the SDL library. By using TObjStream.RegisterDefaultClasses, SuperStream will automatically be able to save and load TObjList objects.

Epilogue

The creation of SDL and SuperStream has been a long, but satisfying journey through the intricacies of building a complex library package, and adapting theories to the realities of a programming environment.

I'd like to take a moment to thank certain individuals whose work has made mine possible:

Stepanov and Lee, the creators of STL, whose architectural work made these kinds of libraries possible, and provided the roadmap for constructing new ones.

ObjectSpace's JGL team, who adapted STL to Java to create the Java Generic Library, and in doing so, proved that the STL concepts could migrate from one language to another.

Inprise, for creating the Delphi environment. I've heard lately that a programmer's favorite environment becomes his hammer, and everything starts to look like a nail to that person. Well, Delphi is my hammer, and I've pounded a ton of nails with it in the last five years. It is, without a doubt, the most productive environment I've ever worked in, and is a very precise match to my list of wants. Delphi just keeps getting better and better. I hope SDL gives it new credibility.

My friends: Kurt and Melanie Westerfeld, Tim Sheridan, Larry Chang (thanks for the office space), Steve Giordano Jr., Steve Giordano Sr., Tim Shinkle, and Paula Thomson. They delivered well-timed criticisms and encouragement.

The SDL Reviewers:

Xavier Pacheco; William Mann; Robert P Kerr; Robert Marsh; Rob Lafreniere; Ray Konopka; Phillip Woon; Peter Roth; Pablo Pissanetzky; Mark Vaughan; Mark Leymaster; Marco Cantu'; Luk Vermeulen; Louis Kleiman; Kurt Westerfeld; Julian Bucknall; John Elrick; J Merrill; Deven Hickingbotham; Danny Thorpe; Brad Stowers; Josh Dahlby