Rockall-DB

Technical Manual

Rockall-DB Version 16.1
Rockall Software Ltd.
24th March 2016

UK Patent Application Number 1511960.5
US Patent Application Number 15/056,092
E-Mail: MABParkes@gmail.com
Telephone: +44 1384 400058

Important Information

It is not uncommon for computer software to contain bugs, mistakes or other issues that may cause it to malfunction in a variety of unexpected ways. Consequently, Rockall-DB is provided on an 'as-is' basis as these types of problems may be present in Rockall-DB and thereby cause it to malfunction without warning. Although considerable effort has been expended trying to identify any defects in Rockall-DB, it is highly likely that serious defects are still present within the software. Consequently, it is strongly advised that any software that uses Rockall-DB is carefully tested to ensure that it functions as expected and required. No responsibility for damages or any other kinds of losses can be accepted by the owners or licensees of Rockall-DB for any problems related to the product or associated materials. Please see the related license for additional terms and conditions.

Copyright

Contents

1. Introduction

A number of common programming languages (such as ‘C’, C++, C# and Java) allow software developers to dynamically allocate main memory using function calls or syntax similar to the following:

```
MY_TYPE *MemoryAddress = new MY_TYPE;
```

or

```
void *MemoryAddress = malloc( sizeof(MY_TYPE) );
```

Now let’s imagine what software development might be like if files and transactional databases worked in a similar way. In such an environment, the code to allocate some file space might look something like:

```
MY_TYPE *FileAddress = new MY_TYPE;
```

or

```
void *FileAddress = malloc( sizeof(MY_TYPE) );
```

Unfortunately, it is unlikely that any worthwhile solution could ever be that simple, as any code would probably need to know the address where the data was stored in the file (i.e. the file `‘Address’`) and also the address where the data was stored in memory (i.e. the `‘Data’` address). So realistically the code would need to be something more like the following:

```
long long int Address;
void *Data;

if ( New( & Address,& Data ) MY_TYPE )
  { /* Use new file space here */ }
```

Although this might be a small step forward, it would still not fully resolve all the issues associated with file management, as a file could still be easily corrupted if a program crashed while the file was being updated. So in addition, there would need to be some kind of transactional support similar to databases. This would mean that the code would need to be more like the following:

```
if ( BeginTransaction() )
  {
  long long int Address;
  void *Data;

  if ( New( & Address,& Data ) MY_TYPE )
    { /* Use new file space here */ }

  EndTransaction();
  }
```

What we have outlined above is not a dream but something you can actually code using Rockall-DB. A complete executable example of a small program to transactionally write ‘Hello World’ into a file is shown in ‘Example-01-01’ below:

```
#include <stdio.h>
#include <string.h>

#include "DatabaseHeap.hpp"
#include "RockallNew.hpp"
#include "RockallTypes.hpp"

int main( int Count,char *Argument[] )
  {
  DATABASE_HEAP Heap;

  if ( (Count == 2) && (Heap.CreateFile( Argument[1] )) )
    {
    if ( Heap.BeginTransaction() )
      {
      FILE_ADDRESS Address;
      char (*Data)[16];

      if ( ROCKALL_NEW<char[16]>::New( & Heap,& Address,& Data ) )
        { strcpy( ((char*) Data),"Hello world !" ); }

      Heap.EndTransaction();
      }

    Heap.CloseFile();
    }

  return 0;
  }
```

<u>Example-01-01</u>

Now that we have covered the basic concept of Rockall-DB let’s move on and see how it might be used in a more practical example. Let’s take a look at how we could transactionally build a new table along with 2 related indexes. We shall begin by taking a look at the necessary definitions and structures in ‘Example-01-02’ below:

```
#include "DatabaseHeap.hpp"
#include "FixedString.hpp"
#include "NoLock.hpp"
#include "RockallTypes.hpp"
#include "Tree.hpp"

typedef DATABASE_HEAP HEAP_TYPE;
typedef FIXED_STRING<HEAP_TYPE,char,32> STRING;
typedef TREE<HEAP_TYPE,NO_LOCK,int,FILE_ADDRESS> INDEX1;
typedef TREE<HEAP_TYPE,NO_LOCK,STRING,FILE_ADDRESS> INDEX2;

typedef struct
  {
  INDEX1     Index1;
  INDEX2     Index2;
  }
ROOT;

typedef struct
  {
  int        Key1;
  STRING     Key2;
  STRING     Value1;
  STRING    Value2;
  }
ROW;
```

Example-01-02

The purpose of 'Example-01-02' is not to explain how Rockall-DB works but rather to give an overall feel for the type of code that is needed to use it. Hopefully, the above code should be fairly obvious for any developer familiar with languages such as 'C', C++, C# or Java (see chapter 7 for multi-language support). All we are doing in this code is including a few Rockall-DB modules, making a few type definitions and specifying a couple of structures.

We see the remainder of the code in 'Example-01-03' below. This code creates a new transactional database, a new transaction and the new indexes. It then builds both the indexes and adds the all rows using a simple 'for' loop. Finally, it ends the transaction and closes the file.

```
#include <stdio.h>
#include <string.h>

#include "AutomaticHeapScope.hpp"
#include "Example-01-02.hpp"
#include "RockallNew.hpp"

int main( int Count,char *Argument[] )
  {
  HEAP_TYPE Heap;

  if ( (Count == 2) && (Heap.CreateFile( Argument[1] )) )
    {
    AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope( & Heap );
    static const int MaxRows = 5;
    static ROW Rows[ MaxRows] =
      {
      { 1,"One","Value1","Value2" },
      { 2,"Two","Value3","Value4" },
      { 3,"Three","Value5","Value6" },
      { 4,"Four","Value7","Value8" },
      { 5,"Five","Value9","Value10" }
      };

    if ( Heap.BeginTransaction() )
      {
      bool Abort = false;
      FILE_ADDRESS Address;
      ROOT *Root;

      if ( ROCKALL_NEW<ROOT>::New( & Heap,& Address,& Root ) )
        {
        int Count;

        for( Count=0;Count < MaxRows;Count ++ )
          {
          ROW *Row1 = & Rows[ Count ];
          ROW *Row2;

          if
              (
              (ROCKALL_NEW<ROW>::New( & Heap,& Address,& Row2 ))
                &&
              (Root -> Index1.NewKey( Row1 -> Key1,Address ))
                &&
              (Root -> Index2.NewKey( Row1 -> Key2,Address ))
              )
            { (*Row2) = (*Row1);  }
          else
            { Abort = true; }
          }
        }
      else
        { Abort = true; }

      Heap.EndTransaction( Abort );
      }

    Heap.CloseFile();
    }

  return 0;
  }
```

Example-01-03

Again, the purpose of ‘Example-01-03’ is not to explain how Rockall-DB works but to give an overall feel for the product. Hopefully, it can be seen that with the required code in this case is fairly straight-forward. Furthermore, hopefully it is also obvious that the example could easily be extended to include any number of columns, indexes, rows or tables.

In summary, what Rockall-DB introduces is a new concept to software development where the memory allocator also offers all the functionality of a transactional database. This new concept makes the historical boundaries between main memory, files and transactional databases simply melt away and throws open the gates to a dizzying array of new programming paradigms for software developers.

2. What is Rockall-DB?

What we have seen in chapter 1 is a very different paradigm for programming transactional databases. A new methodology that naturally integrates into modern programming languages and removes the need for historical anachronisms like embedded SQL statements. A method that is not only simpler but also much faster because the core interface is based on simple native code procedure calls (see chapter 7 for multi-language support). In Rockall-DB, you can effortlessly obtain amazing levels of performance because there are no longer any bloated overheads associated with accessing transactional data. A single CPU core can often achieve performance levels of around 100,000 transactions per second (tps) for simple transactions. A simple 10 million row table with an index can be built in around 1 minute 15 seconds (using a parallel transaction and multiple cores). Alternately, a huge 1 billion row table with an index can be built in around 3 hours 44 minutes. This is because Rockall-DB is the sports car of transactional databases. As a developer, all you need to do is to push down the pedal, hold on tight and feel the surge of power.

What we have seen so far just scratches the surface of Rockall-DB. There is a rich set of core functionality supporting facilities such as asynchronous I/O, enhancements for database locality, a high performance C++ version of garbage collection, support for multithreaded transactions and remote duplicate copies of 'live' databases updated in real-time (i.e. for data security). Additionally, the Rockall-DB library also provides a variety of pre-built functionality such as hash tables, queues, sets, stacks, strings and trees (i.e. indexes), with or without encryption, with or without locks, which work in main memory or within transactional databases. In Rockall-DB, a developer can construct almost anything from a simple transactional file to column families, document stores, graph databases, hierarchic databases, key value databases, multidimensional databases, object oriented databases, relational databases, wide column stores and XML databases (or any combination of these) within a simple application or a complex system. In short, almost all of the limitations and restrictions of traditional databases simply evaporate when using Rockall-DB.

When considering low end or embedded applications Rockall-DB is almost peerless. Almost no other transactional database has a footprint of under a megabyte (i.e. 1MB). This makes it a wonderful tool for the 'internet of things' (i.e. embedded applications such as cars, cameras, phones) providing all the benefits of a full transactional database in very low end devices.

While Rockall-DB may appear a little simplistic on the surface it is believed that this notion quickly dissipates. The key to Rockall-DB's power is that extends the traditional memory management programming model (i.e. '`new`' and '`delete`') to fully incorporate the functionality of high end transactional databases. This means that the boundary between main memory and permanent storage just melts away. Thus, it becomes trivial to make main memory data formats (i.e. in-memory data) the same as permanent storage data formats (i.e. on-disk data), leading to dramatic coding simplifications and yielding sky-high levels of performance. Clearly, combining these two data formats means that data in storage can be bought directly into main memory and used immediately, without the any need for additional buffering, conversions, copying or any other form of data manipulation. Clearly, the structure of the data can directly match what the program requires as there is no database interface or model to get in the way (i.e. no columns, rows or third normal form to consider). A stored data object can be anything from a simple class (or structure) all the way to complex compound class (or structure) optionally including arrays, unions, inheritance or multiple-inheritance. Furthermore, almost everything in Rockall-DB compiles to native code and so there are no overheads associated with garbage collectors, interpreters or similar barriers to performance. In short, Rockall-DB is a very different type of transactional database that works *with* a developer and not against them.

The heart of Rockall-DB is simple link library along with an associated dynamic link library (i.e. `'RockallDB.lib'` and `'RockallDB.dll'`), so deploying Rockall-DB as part of an application is easy (i.e. there is nothing to install - just a couple of files to copy). A Rockall-DB database is a single operating system file and can be stored and managed just like any other operating system file. It can be put in an e-mail or on a USB stick and easily moved to wherever it is needed. If security is a concern, there is a mechanism so it can be encrypted to any level that is required.

Consequently, Rockall-DB means that developers are no longer forced to use centralize databases but can instead partially or fully decentralize them (i.e. a database per patient for Medical Records (MRs) instead of a centralized database). Continuing this thought, Rockall-DB supports some exotic functionality, such as storing multiple smaller Rockall-DB databases within a single larger database (i.e. nested databases). These types of features open the way to whole host of new models for data management, security and storage. Regardless, Rockall-DB can still also be used in the traditional way to create a traditional huge monolithic multi-terabyte database containing large numbers of tables and indexes if that is required for some reason. In short, Rockall-DB is flexible and scalable from just a few of kilobytes all the way to a few terabytes.

While it is common for transactional databases to use proprietary closed data formats, this is not the situation with Rockall-DB. Unlike most other transactional databases Rockall-DB has an open storage placement policy allows the data to remain visible to developers at all times (unless it is encrypted of course). As an example, if `'New'` is used to allocate some space in a Rockall-DB database then the `'FILE_ADDRESS'` returned is the actual offset (in bytes) of this space from the start of the file. Consequently, it is always possible to know exactly where any piece of data has been stored within a Rockall-DB database. Furthermore, unrelated programs can open a Rockall-DB database and access any piece of stored data *without using any part* of product. Consequently, while a lot of products claim they are 'open' in reality Rockall-DB is one of the few products that actually delivers on this promise in a significant way.

Clearly, it is hard to cover all the facets of Rockall-DB in just a few paragraphs. So in chapter 3 and chapter 4 below we will take a brief technical tour of the Rockall-DB core and the Rockall-DB library respectively to highlight a number of its main features. While some portions of these chapters may occasionally be a little tedious, a quick read through them should be useful when it comes to the more complex functionality discussed later in chapter 5 and chapter 6 later.

3. A Tour of the Rockall-DB Core

The heart of Rockall-DB is a common base class named `'VIRTUAL_HEAP'`. This virtual base class specifies a collection of common functions supported by the entire family of Rockall-DB heaps. The names of the Rockall-DB heaps are `'SINGLE_THREADED_HEAP'`, `'MULTI_THREADED_HEAP'`, `'TRANSACTION_HEAP'` and `'DATABASE_HEAP'`.

The simplest and highest performing of the Rockall-DB heaps is `'SINGLE_THREADED_HEAP'` which supports functionality similar to the functions `'malloc'`, `realloc'` and `'free'` in 'C' or `'new'` and `'delete'` in C++, C# or Java. A `'SINGLE_THREADED_HEAP'` is not thread safe and so cannot be used by more than one thread at the same time. A `'MULTI_THREADED_HEAP'` is essentially the same as a `'SINGLE_THREADED_HEAP'` but is thread safe and so can be used by multiple threads at the same time. A `'TRANSACTION_HEAP'` extends the features of a `'MULTI_THREADED_HEAP'` by including support for 'in-memory' transactions. Finally, the most powerful heap is a `'DATABASE_HEAP'` which extends the features of a `'TRANSACTION_HEAP'` by including full transactional support for files (i.e. a full transactional database).

A Rockall-DB heap is a simple class just like any other class in 'C++', C# or Java (see chapter 7 for multi-language support). Consequently, a new Rockall-DB heap can be created as shown in 'Example-03-01' below:

```
#include "SingleThreadedHeap.hpp"

int main( void )
  {
  SINGLE_THREADED_HEAP Heap;

  return 0;
  }
```

Example-03-01

It is possible to have any number of heaps. A heap may be free standing or placed within a data structure such as an array, structure or union. Consequently, we could specify an array of heaps as shown in 'Example-03-02' below:

```
#include "SingleThreadedHeap.hpp"

int main( void )
  {
  SINGLE_THREADED_HEAP Heap[4];

  return 0;
  }
```

Example-03-02

Alternatively, we might have an array of different types of heaps as shown in 'Example-03-03' below:

```
#include "SingleThreadedHeap.hpp"
#include "MultiThreadedHeap.hpp"
#include "TransactionalHeap.hpp"
```

```
#include "DatabaseHeap.hpp"

int main( void )
  {
  SINGLE_THREADED_HEAP Type1;
  MULTI_THREADED_HEAP Type2;
  TRANSACTIONAL_HEAP Type3;
  DATABASE_HEAP Type4;
  VIRTUAL_HEAP *Heap[4] = { & Type1,& Type2,& Type3,& Type4 };

  return 0;
  }
```

Example-03-03

An allocation can be made on any Rockall-DB heap by calling the function `'New'`. However, we first need to specify a simple class to help illustrate how this works. The simple class we shall create is called `'BASIC_CLASS'` and is shown in 'Example-03-04' below:

```
#include <stdio.h>

class BASIC_CLASS
  {
  public:
    int      Value;

    BASIC_CLASS( void ) : Value(0)
      { printf( "Constructor\n" ); }

    ~BASIC_CLASS( void )
      { printf( "Destructor\n" ); }
  };
```

Example-03-04

Now, given the `'BASIC_CLASS'` outlined above we can see how `'New'` works in 'Example-03-05' below:

```
#include "Example-03-04.hpp"
#include "SingleThreadedHeap.hpp"

int main( void )
  {
  BASIC_CLASS *BasicClass;
  SINGLE_THREADED_HEAP Heap;

  if ( Heap.New( ((void**) & BasicClass),sizeof(BASIC_CLASS) ) )
    { BasicClass -> Value = 0; }

  return 0;
  }
```

Example-03-05

The function `'New'` does not call the constructor like `'new'` in C++, C# or Java. It merely allocates the required amount of space. Obviously, the constructor can be called in a number of ways as will be seen later.

Although we are currently focusing our discussion on a `'SINGLE_THREADED_HEAP'` we should remember that almost everything we are seeing also applies equally to all Rockall-DB heaps. Later, we will see that we can allocate space in a transactional database by calling `'New'` on `'DATABASE_HEAP'` in much the same way as we are currently allocating space in main memory using a `'SINGLE_THREADED_HEAP'`. However, for the time being let's continue our focus on `'SINGLE_THREADED_HEAP'` and simple memory allocations.

It should be noted that all Rockall-DB heaps (except for `'DATABASE_HEAP'`) automatically clean up by calling `'DeleteAll'` when they go out of scope. Consequently, the memory allocation created in 'Example-03-05' above would automatically be deleted when `'return 0'` is reached.

A warning message is automatically generated when using the debugging build of Rockall-DB if all the allocations on a heap are not deleted when it goes out of scope. This can be suppressed by manually calling `'DeleteAll'` at the end of the block as shown in 'Example-03-06' below:

```
#include "Example-03-04.hpp"
#include "SingleThreadedHeap.hpp"

int main( void )
  {
  BASIC_CLASS *BasicClass;
  SINGLE_THREADED_HEAP Heap;

  if ( Heap.New( ((void**) & BasicClass),sizeof(BASIC_CLASS) ) )
    { BasicClass -> Value = 0; }

  Heap.DeleteAll();

  return 0;
  }
```

Example-03-06

Now obviously we need to execute the constructor and destructor in most cases. We can do this is in a number of ways and we will begin by showing how this can be done manually for both `'New'` and `'Delete'` in 'Example-03-07' below:

```
#include "Example-03-04.hpp"
#include "PlacementNew.hpp"
#include "SingleThreadedHeap.hpp"

int main( void )
  {
  BASIC_CLASS *BasicClass;
  SINGLE_THREADED_HEAP Heap;

  if ( Heap.New( ((void**) & BasicClass),sizeof(BASIC_CLASS) ) )
    {
    PLACEMENT_NEW( BasicClass,BASIC_CLASS );

    // 'BASIC_CLASS' ready for use.

    PLACEMENT_DELETE( BasicClass,BASIC_CLASS );
    if ( ! Heap.Delete( BasicClass ) )
      { printf( "Delete fails\n" ); }
    }

  return 0;
  }
```

Example-03-07

Clearly, manually calling the constructor and destructor in this way is not ideal and so there is a more automatic way of doing it. Unfortunately, this is not as straightforward as might be hoped. It can be seen that the heap's `'Delete'` function returns a `'BOOLEAN'` value that could be `'False'` (i.e. if the address supplied in `'BasicClass'` was not currently allocated on the `'Heap'`). Obviously, it is much better to return `'False'` than simply crash (i.e. like some heaps), but this does pose a few extra problems.

Now, if `'Delete'` did return `'False'` in this example we would have already executed the destructor and this may have damaged an area of memory that was not allocated on the `'Heap'`. Obviously, this would most likely be a fatal error and most programs would need to terminate.

Unfortunately, C++, C# and Java were designed on the premise of a single heap and in the belief that calls to `'delete'` would never need parameters and could never fail. Consequently, although it would be wonderful to use the built in `'new'` and `'delete'` operators it is necessary in Rockall-DB to use the alternatives shown in 'Example-03-08' below:

```
#include "Example-03-04.hpp"
#include "RockallDelete.hpp"
#include "RockallNew.hpp"
#include "SingleThreadedHeap.hpp"

int main( void )
  {
  BASIC_CLASS *BasicClass;
  SINGLE_THREADED_HEAP Heap;

  if ( ROCKALL_NEW<BASIC_CLASS>::New( & Heap,& BasicClass ) )
    {
    // 'BASIC_CLASS' ready for use.

    if ( ! ROCKALL_DELETE<BASIC_CLASS>::Delete( & Heap,BasicClass ) )
      { printf( "Delete fails\n" ); }

    }

  return 0;
  }
```

Example-03-08

The `'ROCKALL_NEW'` and `'ROCKALL_DELETE'` classes return a `'BOOLEAN'` and require the target type to be supplied as a template parameter. It is also necessary to supply a pointer to the desired `'Heap'` and an appropriately typed instance pointer. Although this is a little different from the traditional `'new'` and `'delete'` operators hopefully, it is close enough to be comfortable for most developers.

Again, it should be remembered that the functioning of the `'ROCKALL_NEW'` and `'ROCKALL_DELETE'` classes also apply to a `'DATABASE_HEAP'` and so constructors and destructors would also execute when new allocations were made or deleted in a transactional database. This is little novel and is something that might not be immediately expected.

Now, it might be noticed that 'Example-03-08' does not help in the situation where `'DeleteAll'` is used as all the memory allocations would still be deleted without calling any destructors.

This can be easily overcome but first we will need to extend the `'BASIC_CLASS'` shown in 'Example-03-04' and create a new `'EXTENDED_CLASS'` as shown in 'Example-03-09' below:

```
#include <stdio.h>
#include "VirtualDestructor.hpp"

class EXTENDED_CLASS : public VIRTUAL_DESTRUCTOR
  {
  public:
    int     Value;

    EXTENDED_CLASS( void ) : Value(0)
      { printf( "Constructor\n" ); }

    ~EXTENDED_CLASS( void )
      { printf( "Destructor\n" ); }
  };
```

Example-03-09

We can see that the difference between the `'BASIC_CLASS'` in 'Example-03-04' and the `'EXTENDED_CLASS'` in 'Example-03-09' is that the `'EXTENDED_CLASS'` inherits from a new class called `'VIRTUAL_DESTRUCTOR'`.

The `'VIRTUAL_DESTRUCTOR'` class simply ensures that there is a 'v-table' associated with an object and that the first entry in the 'v-table' pointers to its destructor. We can now use this 'v-table' to automatically execute the related destructors as shown in 'Example-03-10' below:

```
#include "Example-03-09.hpp"
#include "RockallNew.hpp"
#include "SingleThreadedHeap.hpp"
#include "RockallTypes.hpp"

int main( void )
  {
  EXTENDED_CLASS *ExtendedClass;
  SINGLE_THREADED_HEAP Heap;

  if
      (
      ROCKALL_NEW<EXTENDED_CLASS>::New
        ( & Heap,& ExtendedClass,1,NULL,True )
      )
```

```
    {
    // 'EXTENDED_CLASS' ready for use.
    }

  return 0;
  }
```

Example-03-10

There are a number of modifications in ‘Example-03-10’ that are worth highlighting. The call to `‘New’` has gained three extra parameters. The first and second are `‘1’` and `‘NULL’` respectively and these can usually be skipped as they are the default values. The third is the `‘Destroy’` parameter which is set to `‘True’`. When `‘Destroy’` is `‘True’` all Rockall-DB heaps (except `‘DATABASE_HEAP’`) assume the class has been inherited from the `‘VIRTUAL_DESTRUCTOR’` class and will automatically call the associated destructor via the ‘v-table’ when the a memory allocation is eventually deleted. In this case, the deletion will occur when `‘return 0’` is reached and `‘SINGLE_THREADED_HEAP’` goes out of scope (as discussed in ‘Example-03-05’ and ‘Example-03-06’ above).

A memory allocation created with `‘Destroy’` is `‘True’` can also be deleted manually in a number of ways. A simple `‘Delete’` can be used as shown in ‘Example-03-11’ below:

```
#include "Example-03-09.hpp"
#include "RockallNew.hpp"
#include "SingleThreadedHeap.hpp"
#include "RockallTypes.hpp"

int main( void )
  {
  EXTENDED_CLASS *ExtendedClass;
  SINGLE_THREADED_HEAP Heap;

  if
      (
      ROCKALL_NEW<EXTENDED_CLASS>::New
        ( & Heap,& ExtendedClass,1,NULL,True )
      )
    {
    // 'EXTENDED_CLASS' ready for use.

    if ( ! Heap.Delete( ExtendedClass ) )
      { printf( "Delete fails\n" ); }
    }

  return 0;
  }
```

Example-03-11

We noted in ‘Example-03-07’ that calling `‘Delete’` did not call the destructor. However, we now see in ‘Example-03-11’ that when `‘Destroy’` is `‘True’` in the call to `‘New’` that the destructor will now be called automatically by `‘Delete’`.

In some ways, this is better method than we have seen above as the destructor is only called after the allocation has been found in the `'Heap'` and so the destructor will only be executed if necessary.

We can also use the `'ROCKALL_DELETE'` class as shown in 'Example-03-12' below:

```
#include "Example-03-09.hpp"
#include "RockallDelete.hpp"
#include "RockallNew.hpp"
#include "SingleThreadedHeap.hpp"
#include "RockallTypes.hpp"

int main( void )
  {
  EXTENDED_CLASS *ExtendedClass;
  SINGLE_THREADED_HEAP Heap;

  if
      (
      ROCKALL_NEW<EXTENDED_CLASS>::New
        ( & Heap,& ExtendedClass,1,NULL,True )
      )
    {
    // 'EXTENDED_CLASS' ready for use.

    if
        (
        ! ROCKALL_DELETE<EXTENDED_CLASS>::Delete
          ( & Heap,ExtendedClass )
        )
      { printf( "Delete fails\n" ); }
    }

  return 0;
  }
```

Example-03-12

What is interesting in this case is the management of the destructor. Clearly, the destructor would be manually called in the `'ROCKALL_DELETE'` class before calling `'Delete'`, just like in 'Example-03-07'. Furthermore, the destructor would also normally be called again when `'Delete'` was executed as `'Destroy'` was set to `'True'`. However, the `'Delete'` function supports the `'NoDestroy'` parameter which suppresses the execution of the destructor if set to `'True'`. Consequently, the `'NoDestroy'` parameter is used in `'ROCKALL_DELETE'` to ensure that the destructor is not executed for a second time.

We can also use `'DeleteAll'` as shown in 'Example-03-13' below:

```
#include "Example-03-09.hpp"
#include "RockallNew.hpp"
#include "SingleThreadedHeap.hpp"
#include "RockallTypes.hpp"

int main( void )
  {
  EXTENDED_CLASS *ExtendedClass;
  SINGLE_THREADED_HEAP Heap;
  if
      (
      ROCKALL_NEW<EXTENDED_CLASS>::New
        ( & Heap,& ExtendedClass,1,NULL,True )
      )
    {
    // 'EXTENDED_CLASS' ready for use.
    }

  Heap.DeleteAll();

  return 0;
  }
```

<u>Example-03-13</u>

The `'DeleteAll'` function is another interesting case. When a number of memory allocations are made with `'Destroy'` set to `'True'` a subsequent call to `'DeleteAll'` will first call all the destructors on allocations where `'Destroy'` was set to `'True'` (in an efficient but undefined order) and will then delete all of the memory allocations on the heap. This can be thought of as an efficient C++ form of garbage collection.

Consequently, an optional programming model for Rockall-DB is to create a number of heaps for storing memory allocations with different life times. All (or some) of the memory allocations could be created with `'Destroy'` set to `'True'`. When one of these heaps was no longer needed a simple call to `'DeleteAll'` would efficiently and reliably delete all the related allocations in a single step.

There are a number of significant advantages of such a memory management model. The most obvious is the improvement in performance over traditional garbage collection, as there is no need to track any memory pointers. Another key advantage is that precisely the right amount of memory can be deleted at precisely the right points during execution (i.e. say at the end of a transaction). Finally, there is no need to worry about memory leaks or calling `'Delete'` (except where this is easy, efficient or helpful) as a single well-placed call to `'DeleteAll'` can be used as a 'catch all' to clean up.

We have initially just considered the functionality of the non-transactional heaps, which are `'SINGLE_THREADED_HEAP'` and `'MULTI_THREADED_HEAP'`. Let's now move on and take a look at what can be done with the transactional heaps, which are `'TRANSACTIONAL_HEAP'` and `'DATABASE_HEAP'`. These latter heaps require an active transaction before calling functions such as `'New'` and `'Delete'`, which was not previously necessary.

Let's start by rewriting 'Example-03-08' to use `'DATABASE_HEAP'` instead of a `'SINGLE_THREADED_HEAP'` in 'Example-03-14' below:

```
#include "DatabaseHeap.hpp"
#include "Example-03-04.hpp"
#include "RockallDelete.hpp"
#include "RockallNew.hpp"
```

```
int main( int Count,char *Argument[] )
  {
  DATABASE_HEAP Heap;

  if ( (Count == 2) && (Heap.CreateFile( Argument[1] )) )
    {
    if ( Heap.BeginTransaction() )
      {
      BASIC_CLASS *BasicClass;

      if ( ROCKALL_NEW<BASIC_CLASS>::New( & Heap,& BasicClass ) )
        {
        // 'BASIC_CLASS' ready for use.

        ROCKALL_DELETE<BASIC_CLASS>::Delete( & Heap,BasicClass );
        }

      Heap.EndTransaction();
      }

    Heap.CloseFile();
    }

  return 0;
  }
```

Example-03-14

The main difference between ‘Example-03-08’ and ‘Example-03-14’ is the call to `‘CreateFile’` and the corresponding call to `‘CloseFile’` and the call to `‘BeginTransaction’` and the corresponding call to `‘EndTransaction’`.

Now, as we mentioned previously a `‘DATABASE_HEAP’` supports full transactional database functionality. Consequently, it is necessary to create a new database file by first calling `‘CreateFile’` and finally close it by calling `‘CloseFile’`. Likewise, we need to create a transaction scope by calling `‘BeginTransaction’` before we can call `‘New’` or `‘Delete’` and later terminate it by calling `‘EndTransaction’`. Excluding these changes, the remainder of the code is very similar to ‘Example-03-08’.

We note in ‘Example-03-14’ that we create a new database file, make an allocation in it, immediately delete the new allocation and finally close the file. Clearly, this is not a very illuminating example. Consequently, let’s now look at something a little more realistic in ‘Example-03-15’ below:

```
#include "DatabaseHeap.hpp"
#include "Example-03-04.hpp"
#include "RockallDelete.hpp"
#include "RockallNew.hpp"

int main( int Count,char *Argument[] )
  {
  DATABASE_HEAP Heap;

  if ( (Count == 2) && (Heap.CreateFile( Argument[1] )) )
    {
    if ( Heap.BeginTransaction() )
      {
      BASIC_CLASS *BasicClass;
```

```
        if ( ROCKALL_NEW<BASIC_CLASS>::New( & Heap,& BasicClass ) )
          { BasicClass -> Value = 1; }

        Heap.EndTransaction();
        }

      Heap.CloseFile();
      }

    return 0;
    }
```

Example-03-15

We see in ‘Example-03-15’ that we create a new database file using the name supplied in `'Argument[1]'` and allocate a new `'BasicClass'` object in the file. Once we have successfully created the new `'BasicClass'` object we then set the `'Value'` member to `'1'`.

We have a bit of a subtle problem in this example, as we have no idea where the new `'BasicClass'` object is stored in the file. Moreover, even if we did it would not help as we would probably lose this information when the program reached the `'return 0'` statement.

We can resolve these issues by using some additional features of Rockall-DB as shown in ‘Example-03-16’ below:

```
#include "DatabaseHeap.hpp"
#include "Example-03-04.hpp"
#include "RockallDelete.hpp"
#include "RockallNew.hpp"
#include "RockallTypes.hpp"

int main( int Count,char *Argument[] )
  {
  DATABASE_HEAP Heap;

  if ( (Count == 2) && (Heap.CreateFile( Argument[1] )) )
    {
    if ( Heap.BeginTransaction() )
      {
      BASIC_CLASS *BasicClass;
      FILE_ADDRESS Address;

      if
            (
            (ROCKALL_NEW<BASIC_CLASS>::
                New( & Heap,& Address,& BasicClass ))
              &&
            (Heap.SetUserValue( 0,Address ))
            )
          {
          BasicClass -> Value = 1;

          Heap.EndTransaction( False );
          }
        else
          { Heap.EndTransaction( True ); }
      }

    Heap.CloseFile();
    }

  return 0;
  }
```

Example-03-16

Now, the first significant change we see is an extra parameter on the call to `'New'` called `'Address'`. The `'Address'` parameter is available for all Rockall-DB heaps but its value is always matches to value of the following `'Data'` parameter except for a `'DATABASE_HEAP'` where it is the offset (in bytes) of the new allocation in the file.

Now that we have the `'Address'` of the new allocation in the file we need to save it somewhere so we can find it later. The `'SetUserValue'` function is available in a `'DATABASE_HEAP'` and can used to transactionally save the `'Address'` to a special reserved area in the file. We will see later that when we need the `'Address'` again we can call the `'GetUserValue'` function to retrieve it.

The `'Abort'` parameter is the first parameter to the `'EndTransaction'` function. In 'Example-03-16' we see that if everything works as expected that `'Abort'` is set to `'False'` but if there are problems `'Abort'` is set to `'True'`. When `'Abort'` is set to `'True'` the call to `'EndTransaction'` will undo all calls between the matching `'BeginTransaction'` and `'EndTransaction'` to the functions `'Delete'`, `'New'`, `'Resize'`, `'SetUserValue'`, `and` `'Update'`. In 'Example-03-16' above, this would undo all of the changes that have been made.

Now we have created a simple database with Rockall-DB. Let's take a look at how we could write a program read the data that was stored in it. Again, we will be using some of the additional features of Rockall-DB as shown in 'Example-03-17' below:

```
#include <stdio.h>

#include "DatabaseHeap.hpp"
#include "Example-03-04.hpp"
#include "RockallDelete.hpp"
#include "RockallNew.hpp"
#include "RockallTypes.hpp"

int main( int Count,char *Argument[] )
  {
  DATABASE_HEAP Heap;

  if ( (Count == 2) && (Heap.OpenFile( Argument[1] )) )
    {
    if ( Heap.BeginTransaction() )
      {
      BASIC_CLASS *BasicClass;
      FILE_ADDRESS Address;

      if
          (
          (Heap.GetUserValue( 0,& Address ))
            &&
          (Heap.View( Address,((VOID**) & BasicClass) ))
          )
        { printf( "The value is %d\n",BasicClass -> Value ); }

      Heap.EndTransaction();
      }

    Heap.CloseFile();
    }

  return 0;
  }
```

Example-03-17

Now, the first significant change we see is a call to the `'OpenFile'` function to reopen the file we created in the 'Example-03-16' above. Next, we see a call to the `'GetUserValue'` function to reclaim the `'Address'` we saved using `'SetUserValue'` in the previous example. Next, we call the `'View'` function to load the allocation back into memory from the file and return a pointer to it in `'BasicClass'`. We can now access the data stored in the previous example via the returned pointer in the usual way.

There are a number of calls we could have used instead of the `'View'` function, such as `'ExclusiveView'` and `'Update'`. An `'ExclusiveView'` and a `'View'` are similar in that they permit an area allocated by `'New'` to be examined (i.e. read only) while holding a lock on it. In the case of `'ExclusiveView'` it is a mutually exclusive lock whereas in the case of `'View'` it is a sharable lock. A call to `'Update'` claims an exclusive lock like `'ExclusiveView'` but also takes a copy of the entire area allocated by `'New'`, so that the any changes to the area can be reversed if necessary. After any changes have been made they can be committed by a suitable call to `'EndTransaction'` (i.e. a call without `'Abort'` set to `'True'`).

A call to `'ExclusiveView'` can optionally be terminated by a call to `'EndExclusiveView'`. Also, a call to `'View'` can optionally be terminated by a call to `'EndView'`. Regardless, a call to `'EndTransaction'` will terminate the transaction and terminate all outstanding calls to `'ExclusiveView'`, `'Update'` and `'View'`.

The `'ExclusiveView'`, `'Update'` and `'View'` functions use sophisticated database oriented locks which incorporate 'Automatic Deadlock Detection'. Consequently, all of these functions may legitimately return `'False'` for reasons such as an invalid `'Address'`, a deadlock, a file transfer error or some other issue. Consequently, a developer must always have a strategy for recovering if this occurs. Typically, this is not hard and typically involves aborting the current transaction and simply trying again.

A `'TRANSACTIONAL_HEAP'` is very similar a `'DATABASE_HEAP'` in most respects except that all of the operations are carried out in main memory rather than in a file. Additionally, like the non-transactional heaps `'SINGLE_THREADED_HEAP'` and a `'MULTI_THREADED_HEAP'` it will also automatically call `'DeleteAll'` and clean up when it goes out of scope.

In closing, we have quickly skimmed over some of the main features of the Rockall-DB core. A full description of all of the functions mentioned in this section along with a number of larger examples is available in chapter 5 below. We have also skipped over a few of the more advanced Rockall-DB core features, such as asynchronous I/O, data locality, parallelism and shadow files. These are also covered in in chapter 5 below.

Next, we shall take a tour of the Rockall-DB library. The Rockall-DB library contains a significant quantity of pre-built functionality, such as hash tables, queues, sets, stacks, strings and trees. Furthermore, the Rockall-DB library is supplied in source form and so it is sometimes useful as a source of helpful ideas, methodologies and techniques.

4. A Tour of the Rockall-DB Library

A significant amount of the benefit provided by Rockall-DB is derived from the Rockall-DB library. The Rockall-DB library is a collection of well-known data structures that work with all of the Rockall-DB heaps. The Rockall-DB library is supplied as source form (i.e. C++) and so can be easily analyzed, modified and optimized as required. The main restriction on the Rockall-DB library (or a derivative work) is that any use of it constitutes use of the product and so must comply with the associated product licensing terms.

The Rockall-DB library contains classes for hash tables, queues, sets, stacks, strings and trees (i.e. indexes). All of these classes are C++ templates and are typically automatically optimized by the C++ compiler for specific heaps, types of locking and data types. Moreover, there are some very powerful and unusual features available in the Rockall-DB library, such as massive multi-way trees for multi-terabyte databases (i.e. including up to 16k branching factors per node). Nonetheless, just to keep things simple let's begin with the straight-forward stack oriented functionality as shown in 'Example-04-01' below:

```
#include "AutomaticHeapScope.hpp"
#include "DatabaseHeap.hpp"
#include "Failure.hpp"
#include "Stack.hpp"

static void PushAndPop( void )
  {
  STACK<DATABASE_HEAP,NO_LOCK,int> Stack;
  static const int MaxCycles = 100;
  int Count;
  int Value;

  for
    (
    Count=0;
    (Count < MaxCycles) && (Stack.PushValue( Count ));
    Count ++
    );

  if ( Count < MaxCycles )
    { FAILURE( "Unable to push all the values" ); }

  for
    (
    /* void */;
    (Count > 0) && (Stack.PopValue( & Value ));
    Count --
    );

  if ( Count > 0 )
    { FAILURE( "Unable to pop all the values" ); }
  }

int main( int Count,char *Argument[] )
  {
  DATABASE_HEAP Heap;
  AUTOMATIC_HEAP_SCOPE<DATABASE_HEAP> Scope( & Heap );
  if ( (Count == 2) && (Heap.CreateFile( Argument[1] )) )
    {
    PushAndPop();

    Heap.CloseFile();
    }

  return 0;
  }
```

Example-04-01

We can see in 'Example-04-01' that the code creates a new database and then transactionally pushes and pops 100 integers on and off a stack. It may be noted that the code is very similar to what would normally be written for an in-memory stack. Regardless, there are a few differences which we will cover below.

The first significant difference is in the use of the `'STACK'` class. The `'STACK'` class takes three template parameters. The first template parameter specifies type of heap to be used and would normally be `'SINGLE_THREADED_HEAP'`, `'MULTI_THREADED_HEAP'`, `'TRANSACTIONAL_HEAP'` or `'DATABASE_HEAP'`. This first template parameter allows the resulting object code to be automatically optimized by the C++ compiler for the selected heap. Alternately, `'VIRTUAL_HEAP'` may be used if the type of heap is unknown or if different types of heap are used in different instances of the object. The second template parameter specifies the type of locking to be used and would normally be `'NO_LOCK'`, `'MEMORY_LOCK'` or `'FILE_LOCK'`. All of these lock types can be used in conjunction with any of the Rockall-DB heaps except for `'MEMORY_LOCK'` which cannot be used in conjunction with a `'DATABASE_HEAP'` (as obviously an in-memory lock cannot be stored in a transactional database). Finally, the third template parameter specifies the type of the object to be stored in the `'STACK'`. We see in 'Example-04-01' that it is an `'int'` but it could be any class, structure or type that has a suitable assignment operator (i.e. a `'double'`, a `'FIXED_STRING'`, a `'FLEXIBLE_STRING'`, a `'float'`, a `'long long'` and so on).

A call to `'PushValue'` is used to push a new value onto the `'STACK'` and a call to `'PopValue'` is used to pop a value off the `'STACK'` respectively. Hopefully, this is straight-forward and so needs little explanation.

The next significant difference is the use of the `'AUTOMATIC_HEAP_SCOPE'` class. All of the Rockall-DB data structures (i.e. `'FIXED_STRING'`, `'FLEXIBLE_STRING'`, `'HASH'`, `'QUEUE'`, `'ROW_SET'`, `'SET'`, `'STACK'` and `'TREE'`) require at least one active instance of the `'AUTOMATIC_HEAP_SCOPE'` class on the stack in order to function correctly.

The `'AUTOMATIC_HEAP_SCOPE'` class provides vital information about the heap to be used to the related classes. This information cannot be stored in the class directly as it contains memory pointers and all of The Rockall-DB library classes can be written to file (i.e. when using a `'DATABASE_HEAP'`). Furthermore, this information cannot be consistently passed as parameters as destructors in C++ are not permitted to have parameters. Consequently, it is there is no alternative to managing this information in the `'AUTOMATIC_HEAP_SCOPE'` class, which we will discuss further below.

All of the heaps in Rockall-DB are derived from the base class `'VIRTUAL_HEAP'`. The `'AUTOMATIC_HEAP_SCOPE'` class supports the exactly same interface as `'VIRTUAL_HEAP'` and in most cases an instance can be used as a direct substitute for any of the Rockall-DB heaps. The `'AUTOMATIC_HEAP_SCOPE'` class takes a template parameter which specifies type of heap to be used and would normally be `'SINGLE_THREADED_HEAP'`, `'MULTI_THREADED_HEAP'`, `'TRANSACTIONAL_HEAP'` or `'DATABASE_HEAP'` just like we saw previously with `'STACK'`. Again, this template parameter allows the resulting object code to be automatically optimized by the C++ compiler specifically for the selected type of heap. Also, `'VIRTUAL_HEAP'` may be used if the type of heap is unknown or if different types of heap are to be used in different instances of the related objects.

Now, let's have a closer look at the `'AUTOMATIC_HEAP_SCOPE'` class to better understand how it works. We will modify 'Example-04-01' to create a new 'Example-04-02' below:

```
#include "AutomaticHeapScope.hpp"
#include "DatabaseHeap.hpp"
#include "Failure.hpp"
#include "Stack.hpp"

static void PushAndPop( void )
  {
  AUTOMATIC_HEAP_SCOPE<DATABASE_HEAP> Scope;

  if ( Scope.BeginTransaction() )
    {
    STACK<DATABASE_HEAP,NO_LOCK,int> Stack( & Scope );
    static const int MaxCycles = 100;
    int Count;
    int Value;

    for
      (
      Count=0;
      (Count < MaxCycles) && (Stack.PushValue( & Scope,Count ));
      Count ++
      );

    if ( Count < MaxCycles )
      { FAILURE( "Unable to push all the values" ); }

    for
      (
      /* void */;
      (Count > 0) && (Stack.PopValue( & Scope,& Value ));
      Count --
      );

    if ( Count > 0 )
      { FAILURE( "Unable to pop all the values" ); }
    }
  }

int main( int Count,char *Argument[] )
  {
```

```
DATABASE_HEAP Heap;
AUTOMATIC_HEAP_SCOPE<DATABASE_HEAP> Scope( & Heap );

if ( (Count == 2) && (Scope.CreateFile( Argument[1] )) )
  {
  PushAndPop();

  Scope.CloseFile();
  }

return 0;
}
```

Example-04-02

The first change we note in 'Example-04-02' is the addition of a new `'AUTOMATIC_HEAP_SCOPE'` instance called `'Scope2'`. While it would have been more efficient to simply pass a pointer to `'Scope1'` as a parameter to the `'PushAndPop'` function we have coded the example this way to highlight an important point.

The new instance `'Scope2'` will automatically inherit all its initial values from the nearest matching instance of `'AUTOMATIC_HEAP_SCOPE'` (in this case `'Scope1'`) unless new values are supplied in its constructor. As no new values are supplied in this case `'Scope2'` it will inherit all its values from `'Scope1'` and so `'Scope2'` will be functionally identical to `'Scope1'`. Consequently, a call `'Scope1.GetHeap()'` will return the same result as a call `'Scope2.GetHeap()'` even though on the surface these two instances appear to be unrelated. This is the mechanism most of the Rockall-DB library uses to find the heap for Rockall-DB library classes such as `'FIXED_STRING'`, `'FLEXIBLE_STRING'`, `'HASH'`, `'QUEUE'`, `'ROW_SET'`, `'SET'`, `'STACK'` and `'TREE'`.

The next significant change is the addition of a new call to `'Scope2.BeginTransaction()'`. Now, in 'Example-04-01' we did not call `'BeginTransaction'` and so the `'STACK'` class would have been forced to automatically make calls to `'BeginTransaction'` and `'EndTransaction'` within each call of `'PushValue'` and `'PopValue'`. Consequently, as there are 100 calls of each function, the `'STACK'` class would have created 200 separate transactions (i.e. calls to `'BeginTransaction'` and `'EndTransaction'`). Obviously, this is not very efficient. Now, by making a single call to `'Scope2.BeginTransaction()'` at the outer scope this will reduce the total number of transactions from 200 to 1. This significant reduction occurs because it is no longer necessary to automatically create any new transactions within the `'PushValue'` and `'PopValue'` calls, as one already exists.

It may be noticed that the `'FastPushAndPop'` function does not contain a call to `'EndTransaction'`. This is not necessary as `'AUTOMATIC_HEAP_SCOPE'` automatically calls `'EndTransaction'` if there is an unmatched call to `'BeginTransaction'` when it goes out of scope. A call to `'EndTransaction'` can be made at any point within an `'AUTOMATIC_HEAP_SCOPE'` but this is an option rather than being a necessity.

Finally, it may have be noticed that the calls to the constructor for `'Stack'` and the functions `'PushValue'` and `'PopValue'` are now been passed `'Scope2'` as the first parameter. This is an optimization as it saves these calls from internally creating a third instance of `'AUTOMATIC_HEAP_SCOPE'` just so they can call `'GetHeap'` to access the related heap. Again, this is an option and is only done to improve efficiency and performance.

Now, let's go a step further and show how a single piece of generic code can be written that supports all of the Rockall-DB heaps. We will modify 'Example-04-02' to create a new 'Example-04-03' below:

```
#include "AutomaticHeapScope.hpp"
#include "DatabaseHeap.hpp"
#include "MultiThreadedHeap.hpp"
#include "Failure.hpp"
#include "SingleThreadedHeap.hpp"
#include "Stack.hpp"
#include "TransactionalHeap.hpp"

static void PushAndPop( void )
  {
  AUTOMATIC_HEAP_SCOPE<VIRTUAL_HEAP> Scope;

  if ( Scope.BeginTransaction() )
    {
    STACK<VIRTUAL_HEAP,NO_LOCK,int> Stack( & Scope );
    static const int MaxCycles = 100;
    int Count;
    int Value;

    for
      (
      Count=0;
      (Count < MaxCycles) && (Stack.PushValue( & Scope,Count ));
      Count ++
      );

    if ( Count < MaxCycles )
      { FAILURE( "Unable to push all the values" ); }

    for
      (
      /* void */;
      (Count > 0) && (Stack.PopValue( & Scope,& Value ));
      Count --
      );

    if ( Count > 0 )
      { FAILURE( "Unable to pop all the values" ); }
    }
  }

int main( int Count,char *Argument[] )
  {
  if ( Count == 2 )
    {
    SINGLE_THREADED_HEAP Heap1;
    MULTI_THREADED_HEAP Heap2;
    TRANSACTIONAL_HEAP Heap3;
    DATABASE_HEAP Heap4;
    VIRTUAL_HEAP *Heaps[4] = { & Heap1,& Heap2,& Heap3,& Heap4 };
    int Count;

    for ( Count=0;Count < 4;Count ++ )
      {
      AUTOMATIC_HEAP_SCOPE<VIRTUAL_HEAP> Scope( Heaps[ Count ] );

      if ( Scope.CreateFile( Argument[1] ) )
        {
        PushAndPop();

        Scope.CloseFile();
        }
      }
    }

  return 0;
  }
```

Example-04-03

What we see in 'Example-04-03' is that any code based on `'AUTOMATIC_HEAP_SCOPE'` and designed to work with a `'DATABASE_HEAP'` will also work with all the other types of Rockall-DB heap (i.e. `'SINGLE_THREADED_HEAP'`, `'MULTI_THREADED_HEAP'` and `'TRANSACTIONAL_HEAP')`. This feature is one of the cornerstones of the Rockall-DB library. What happens is that all the unnecessary calls are automatically ignored on by heaps that do not support these functions.

Clearly, it would be reasonable to question the performance of any code written in this way. Nonetheless, we will see later that almost all of the related inefficiencies are usually optimized away by the C++ compiler. The key to all of these optimizations is another feature of the `'AUTOMATIC_HEAP_SCOPE'` class and is discussed below.

We can see that the function `'PushAndPop'` has hardly changed from 'Example-04-02' except for the references to `'DATABASE_HEAP'` have changed to the more generic `'VIRTUAL_HEAP'`. Additionally, the function `'main'` has been modified to include all four types of heap and a loop has been added to call `'PushAndPop'` for each of them in turn. Finally, the `'AUTOMATIC_HEAP_SCOPE'` template has also been changed from `'DATABASE_HEAP'` to the more generic `'VIRTUAL_HEAP'`.

Clearly, a call to functions such as `'CreateFile'`, `'ExclusiveView'`, `'Update'`, `'View'` and `'CloseFile'` should fail on a `'SINGLE_THREADED_HEAP'` or `'MULTI_THREADED_HEAP'` as they have no meaning and are not implemented in these heaps. However, the `'AUTOMATIC_HEAP_SCOPE'` class simply skips these calls (or optimizes them away in most situations) for a `'SINGLE_THREADED_HEAP'` or `'MULTI_THREADED_HEAP'` but retains them for a `'TRANSACTIONAL_HEAP'` or a `'DATABASE_HEAP'`. Consequently, when using the `'AUTOMATIC_HEAP_SCOPE'` class it is 'as if' all Rockall-DB heaps supported the same functionality. Now, this illusion is very shallow as all the unsupported calls simply do nothing but still return `'True'`. Although trivial, this simple mechanism it is powerful enough to permit generic code to be written in Rockall-DB in most situations.

An analysis of the Rockall-DB library will show that this methodology is extensively used to make the code in the library generic. Now, it might be argued that even a small amount of additional code could impact performance and so be undesirable. However, a closer inspection will show that the C++ compiler typically optimizes all the additional code to nothing and so effectively removes it. Let's consider the case of `'NO_LOCK'` from the Rockall-DB library in 'Example-04-04' below:

```
#include "BaseTypes.hpp"
#include "ReservedWords.hpp"
#include "RockallTypes.hpp"

class NO_LOCK
  {
  public:
    NO_LOCK( VOID )
      { /* void */ }

    BOOLEAN ClaimExclusiveLock( BOOLEAN Wait = True )
      { return True; }

    BOOLEAN ClaimSharedLock( BOOLEAN Wait = True )
      { return True; }

    VOID ReleaseExclusiveLock( VOID )
      { /* void */ }

    VOID ReleaseSharedLock( VOID )
      { /* void */ }

    ~NO_LOCK( VOID )
      { /* void */ }
  };
```

Example-04-04

Now, we saw in 'Example-04-03' that the second template parameter to `'STACK'` was related to locking and in this case was passed the value `'NO_LOCK'`. We can now see from 'Example-04-04' that the `'NO_LOCK'` class does not contain any executable code. Consequently, anything but the lowest quality compiler should entirely optimize any reference to it away. We also saw in 'Example-04-03' that the first template parameter to `'STACK'` was the type of heap. Again, the reason for this is to permit the C++ compiler to do similar optimizations, as we shall see from 'Example-04-05' below:

```
#include "AutomaticHeapScope.hpp"
#include "SingleThreadedHeap.hpp"

int main( int Count,char *Argument[] )
  {
  SINGLE_THREADED_HEAP Heap;
  AUTOMATIC_HEAP_SCOPE<SINGLE_THREADED_HEAP> Scope( & Heap );

  if ( (Count == 2) && (Scope.CreateFile( Argument[1] )) )
    {
    if ( Scope.BeginTransaction() )
      { return 1; }
    }
  return 0;
  }
```

Example-04-05

The `'AUTOMATIC_HEAP_SCOPE'` class has been coded to understand the different types of Rockall-DB heaps. The source code in 'Example-04-05' contains multiple 'if' statements. Nonetheless, after C++ compiler optimization phase all that is typically left is shown in 'Example-04-06' below:

```
#include "AutomaticHeapScope.hpp"
#include "SingleThreadedHeap.hpp"

int main( int Count,char *Argument[] )
  {
  SINGLE_THREADED_HEAP Heap;
  AUTOMATIC_HEAP_SCOPE<SINGLE_THREADED_HEAP> Scope( & Heap );

  if ( Count == 2 )
      { return 1; }

  return 0;
  }
```

Example-04-06

The reduction in code occurs because the `'AUTOMATIC_HEAP_SCOPE'` class knows that a `'SINGLE_THREADED_HEAP'` does not implement various functions, such as `'CreateFile'`, `'OpenFile'`, `'BeginTransaction'`, `'ExclusiveView'`, `'EndExclusiveView'`, `'Update'`, `'View'`, `'EndView'`, `'EndTransaction'` and `'CloseFile'`. Consequently, it arranges for these functions to be optimized away in much the same way as `'NO_LOCK'` was optimized away in 'Example-04-04'.

We have now covered the basics of the `'AUTOMATIC_HEAP_SCOPE'` class so let's move on to take a quick look at `'FIXED_STRING'` and `'FLEXIBLE_STRING'` classes. We are only covering these two classes as they help demonstrate some of the more powerful features of the Rockall-DB library. Let's take a quick look at 'Example-04-07' below:

```
#include "AutomaticHeapScope.hpp"
#include "FixedString.hpp"
#include "FlexibleString.hpp"
#include "RockallTypes.hpp"
#include "SingleThreadedHeap.hpp"

typedef SINGLE_THREADED_HEAP HEAP_TYPE;

int main( void )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<SINGLE_THREADED_HEAP> Scope( & Heap );
  FIXED_STRING<SINGLE_THREADED_HEAP,CHAR,64> String1 = "String1";
  FIXED_STRING<SINGLE_THREADED_HEAP,CHAR,64> String2 = "String2";
  FIXED_STRING<SINGLE_THREADED_HEAP,WCHAR,64> String3 = L"String3";
  FIXED_STRING<SINGLE_THREADED_HEAP,WCHAR,64> String4 = L"String4";
  FLEXIBLE_STRING<SINGLE_THREADED_HEAP,CHAR> String5 = "String5";
  FLEXIBLE_STRING<SINGLE_THREADED_HEAP,CHAR> String6 = "String6";
  FLEXIBLE_STRING<SINGLE_THREADED_HEAP,WCHAR> String7 = L"String7";
```

```
FLEXIBLE_STRING<SINGLE_THREADED_HEAP,WCHAR> String8 = L"String8";
CHAR String9[10] = "String9";
WCHAR String10[10] = L"String10";

if ( (String1 != String2) && (String1 != String9) )
  {
  String2 = String1;
  String2 = String3;
  String2 = String5;
  String2 = String7;
  String2 = String9;
  String2 = String10;

  String2 = String1;
  String2 += " & ";
  String2 += String1;
  String2 += " & ";
  String2 += String5;
  String2 += " & ";
  String2 += String9;
  }

if ( (String3 != String4) && (String3 != String10) )
  {
  String4 = String1;
  String4 = String3;
  String4 = String5;
  String4 = String7;
  String4 = String9;
  String4 = String10;

  String4 = String3;
  String4 += L" & ";
  String4 += String3;
  String4 += L" & ";
  String4 += String7;
  String4 += L" & ";
  String4 += String10;
  }

if ( (String5 != String6) && (String5 != String9) )
  {
  String6 = String1;
  String6 = String3;
  String6 = String5;
  String6 = String7;
  String6 = String9;
  String6 = String10;

  String6 = String5;
  String6 += " & ";
  String6 += String1;
  String6 += " & ";
  String6 += String5;
  String6 += " & ";
  String6 += String9;
  }

if ( (String7 != String8) && (String7 != String10) )
  {
  String8 = String1;
```

```
    String8 = String3;
    String8 = String5;
    String8 = String7;
    String8 = String9;
    String8 = String10;

    String8 = String7;
    String8 += L" & ";
    String8 += String3;
    String8 += L" & ";
    String8 += String7;
    String8 += L" & ";
    String8 += String10;
    }

  return 0;
  }
```

Example-04-07

First, we see in ‘Example-04-07’ that strings (i.e. ‘FIXED_STRING’ or ‘FLEXIBLE_STRING’) or constants of the same type can be compared together using the ‘==’, ‘!=’, ‘>’, ‘>=’, ‘<’ and ‘<=’ operators.

Next, we see that the value of a ‘FIXED_STRING’ can be assigned to a ‘FLEXIBLE_STRING’ or vice versa. We also see that a ‘FIXED_STRING’ or ‘FLEXIBLE_STRING’ can be assigned a constant value. Additionally, a value of type ‘CHAR’ can be assigned to a string of type ‘WCHAR’ or vice versa.

Finally, we see that strings (i.e. ‘FIXED_STRING’ or ‘FLEXIBLE_STRING’) or constants of the same type can be joined together using the ‘+=’ operator.

In summary, the Rockall-DB string classes are flexible but fairly basic. They only exist so that they may be used as keys or values in more advanced classes like ‘HASH’, ‘SET’ and ‘TREE’, as these classes require their keys to support the assignment and comparison operators.

Now we understand the basics of ‘FIXED_STRING’ and ‘FLEXIBLE_STRING’ let’s move on to take a look at the ‘HASH’, ‘SET’ and ‘TREE’ classes. We will begin with the ‘HASH’ class but first we need to specify a structure in ‘Example-04-08’ below:

```
#include "FixedString.hpp"
#include "SingleThreadedHeap.hpp"

static const int MaxDetails = 4;
static const int MaxName = 10;

typedef struct
  {
  char      Forename[ MaxName ];
  char      Surname[ MaxName ];
  char      Postcode[ MaxName ];
  }
DETAILS;

static DETAILS Details[ MaxDetails ] =
  {
  { "Albert", "Adams", "SW1 1AA" },
  { "Clive", "Butler", "SW1 1AB" },
  { "Michael", "Jones", "SW1 1AC" },
  { "Jeff", "Smith", "SW1 1AA" }
  };
```

Example-04-08

What we see in ‘Example-04-08’ is a small database of names and postcodes which we will use in later examples. Clearly, in a more realistic application there would be many more names and extra information, such as addresses, phone numbers and so forth. Furthermore, all this information would likely be read in from a file or some other data source. Regardless, just to keep things simple let’s stick to the information provided in ‘Example-04-08’.

Now, let’s consider the code outlined in ‘Example-04-09’ below:

```
#include <stdio.h>

#include "AutomaticHeapScope.hpp"
#include "Example-04-08.hpp"
#include "Failure.hpp"
#include "Hash.hpp"

typedef SINGLE_THREADED_HEAP HEAP_TYPE;
typedef FIXED_STRING<HEAP_TYPE,char,MaxName> NAME;
typedef HASH<HEAP_TYPE,NO_LOCK,NAME,DETAILS*> INDEX;

static bool CreateIndex( INDEX *Index )
  {
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope;
  int Count;

  for ( Count=0;Count < MaxDetails;Count ++ )
    {
    DETAILS *Current = & Details[ Count ];
    NAME Name1 = Current -> Forename;
    NAME Name2 = Current -> Surname;

    if
        (
        (! Index -> NewKey( & Scope,Name1,Current ))
          ||
        (! Index -> NewKey( & Scope,Name2,Current ))
        )
      { FAILURE( "Unable to add key" ); }
    }

  return (Count == MaxDetails);
  }

static void FindKey( char *Name,INDEX *Index )
  {
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope;
  NAME Key = Name;
  DETAILS *Target;

  if ( Index -> FindKey( & Scope,Key,& Target ) )
    {
```

```
    printf
      (
      "The details are: '%s %s' at '%s'\n",
      Target -> Forename,
      Target -> Surname,
      Target -> Postcode
      );
    }
  }

int main( void )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope( & Heap );
  INDEX Index;

  if ( CreateIndex( & Index ) )
    { FindKey( Details[2].Forename,& Index ); }

  return 0;
  }
```

Example-04-09

We see in 'Example-04-09' that we can easily create an in memory `'HASH'` table called `'Index'` that maps `'NAME'` (i.e. a `'FIXED_STRING'`) to a pointer to `'DETAILS'`. We also see later that the values for `'Forename'` and `'Surname'` for all the `'Names'` are added into the `'Index'` using the `'NewKey'` function.

It might be considered a mistake to add both `'Forename'` and `'Surname'` into the same `'Index'`, as a traditional relational database would normally to have one index for `'Forename'` and a second index for `'Surname'`. However, what we are doing here is highlighting the flexibility of Rockall-DB. In some cases, two indexes might be more appropriate and this can easily be implemented by adding a second `'Index'`. Nonetheless, in this case we are considering the situation where customers might calling on the telephone and so we might need to find their records based on the whatever information they have available (which may vary). In this situation, having `'Forename'` and `'Surname'` in the same `'Index'` could be advantageous. Regardless, it is certainly quite a bit easier. Furthermore, we could add extra fields into the `'Index'`, such as telephone numbers, street names or other information, as we will discuss below.

Let's extend 'Example-04-09' to make it a transactional database by using a `'DATABASE_HEAP'` in 'Example-04-10' below:

```
#include <stdio.h>

#include "AutomaticHeapScope.hpp"
#include "AutomaticViewScope.hpp"
#include "Example-04-08.hpp"
#include "Failure.hpp"
#include "Hash.hpp"
#include "RockallNew.hpp"
#include "RockallTypes.hpp"

typedef DATABASE_HEAP HEAP_TYPE;
typedef FIXED_STRING<HEAP_TYPE,char,MaxName> NAME;
typedef HASH<HEAP_TYPE,NO_LOCK,NAME,FILE_ADDRESS> INDEX;

static bool CreateIndex( INDEX *Index )
  {
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope;
  int Count;

  for ( Count=0;Count < MaxDetails;Count ++ )
    {
    DETAILS *Current1 = & Details[ Count ];
    NAME Name1 = Current1 -> Forename;
    NAME Name2 = Current1 -> Surname;
    FILE_ADDRESS Address;
    DETAILS *Current2;

    if
        (
        (ROCKALL_NEW<DETAILS>::New( & Scope,& Address,& Current2 ))
          &&
        (Index -> NewKey( & Scope,Name1,Address ))
          &&
        (Index -> NewKey( & Scope,Name2,Address ))
        )
      { (*Current2) = (*Current1); }
    else
      { FAILURE( "Unable to add key" ); }
    }

  return (Count == MaxDetails);
  }

static void FindKey( char *Name,INDEX *Index )
  {
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope;
  FILE_ADDRESS Address;
  NAME Key = Name;

  if ( Index -> FindKey( & Scope,Key,& Address ) )
    {
    AUTOMATIC_VIEW_SCOPE<HEAP_TYPE> ViewScope( & Scope );
    DETAILS *Target;

    if ( ViewScope.View( Address,((void**) & Target) ) )
      {
      printf
        (
        "The details are: '%s %s' at '%s'\n",
        Target -> Forename,
        Target -> Surname,
        Target -> Postcode
        );
      }
    }
  }

int main( int Count,char *Argument[] )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope( & Heap );

  if ( (Count==2) && (Scope.CreateFile( Argument[1] )) )
```

```
    {
    if ( Scope.BeginTransaction() )
      {
      INDEX Index;

      if ( CreateIndex( & Index ) )
        { FindKey( Details[2].Forename,& Index ); }

      Scope.EndTransaction();
      }

    Scope.CloseFile();
    }

  return 0;
  }
```

Example-04-10

We can see from 'Example-04-10' that using a `'DATABASE_HEAP'` makes things a little more complex. Clearly, we can't store a memory address to `'DETAILS'` in the `'Index'` as memory addresses can't be reliably stored and retieved when using transactional databases. Instead, we need to allocate some new space using `'ROCKALL_NEW'` and then store the returned `'FILE_ADDRESS'` (called `'Address'`) in the `'Index'` in place of the memory address in 'Example-04-09'.

We can see that this also makes using `'FindValue'` a little harder as well. When we call `'FindValue'` we will be returned a `'FILE_ADDRESS'` from the `'Index'` and so we need to call `'View'` to bring the `'DETAILS'` back into memory from the database. In this case, we use the `'AUTOMATIC_VIEW_SCOPE'` class to automatically call the `'EndView'` function at the end of the scope for us, just to make things a bit easier.

Although the code is now a little more complex this does not seem to be disproportionate, considering that we upgraded our example to be a fully transactional database. As an aside, it may be noted that simply changing the `'HEAP_TYPE'` from `'DATABASE_HEAP'` to `'SINGLE_THREADED_HEAP'` would effectively make 'Example-04-10' the same as 'Example-04-09' above. The resulting object code would be also be very similar, as much of the additional code in 'Example-04-10' would simply be optimised away by the C++ compiler as discussed previously.

Now we have upgraded 'Example-04-09' to be a transactional database in 'Example-04-10' let's say that we would like to add the `'Postcode'` into the `'Index'`. This is a little tricky because the `'Postcode'` contains duplicates and none of the Rockall-DB library classes support duplicate keys. Let's take a look at how this is done in 'Example-04-11' below:

```
#include <stdio.h>

#include "AutomaticHeapScope.hpp"
#include "AutomaticViewScope.hpp"
#include "Example-04-08.hpp"
#include "Failure.hpp"
#include "Hash.hpp"
#include "RockallNew.hpp"
```

```
#include "RockallTypes.hpp"
#include "RowSet.hpp"

typedef DATABASE_HEAP HEAP_TYPE;
typedef FIXED_STRING<HEAP_TYPE,char,MaxName> NAME;
typedef HASH<HEAP_TYPE,NO_LOCK,NAME,FILE_ADDRESS> INDEX;
typedef ROW_SET<HEAP_TYPE,NO_LOCK> ROWS;

static bool NewKey( FILE_ADDRESS Address,NAME & Name,INDEX *Index )
  {
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope;
  FILE_ADDRESS Duplicates;
  ROWS *Rows;

  if ( Index -> FindKey( & Scope,Name,& Duplicates ) )
    {
    AUTOMATIC_VIEW_SCOPE<HEAP_TYPE> ViewScope( & Scope );

    return
      (
      ViewScope.View( Duplicates,((void**) & Rows) )
        &&
      Rows -> NewValue( Address )
      );
    }
  else
    {
    FILE_ADDRESS NewRows;

    return
      (
      (ROCKALL_NEW<ROWS>::New( & Scope,& NewRows,& Rows ))
        &&
      Rows -> NewValue( Address )
        &&
      Index -> NewKey( & Scope,Name,NewRows )
      );
    }
  }

static bool CreateIndex( INDEX *Index )
  {
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope;
  int Count;

  for ( Count=0;Count < MaxDetails;Count ++ )
    {
    DETAILS *Current1 = & Details[ Count ];
    NAME Name1 = Current1 -> Forename;
    NAME Name2 = Current1 -> Surname;
    NAME Postcode = Current1 -> Postcode;
    FILE_ADDRESS Address;
    DETAILS *Current2;

    if
        (
        (ROCKALL_NEW<DETAILS>::New( & Scope,& Address,& Current2 ))
          &&
        (NewKey( Address,Name1,Index ))
          &&
        (NewKey( Address,Name2,Index ))
```

```
          &&
        (NewKey( Address,Postcode,Index ))
        )
      { (*Current2) = (*Current1); }
    else
      { FAILURE( "Unable to add key" ); }
    }

  return (Count == MaxDetails);
  }

static void FindKey( char *Name,INDEX *Index )
  {
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope;
  FILE_ADDRESS Address;
  NAME Key = Name;

  if ( Index -> FindKey( & Scope,Key,& Address ) )
    {
    AUTOMATIC_VIEW_SCOPE<HEAP_TYPE> View1( & Scope );
    ROWS *Rows;

    if ( View1.View( Address,((void**) & Rows) ) )
      {
      FILE_ADDRESS *Set;
      FILE_SIZE Size;

      Rows -> Touch( & Scope );

      if ( Rows -> View( & Scope,& Set,& Size ) )
        {
        int Count;

        for ( Count=0;Count < Size;Count ++ )
          {
          AUTOMATIC_VIEW_SCOPE<HEAP_TYPE> View2( & Scope );
          DETAILS *Target;

          if ( View2.View( Set[ Count ],((void**) & Target) ) )
            {
            printf
              (
              "The details are: '%s %s' at '%s'\n",
              Target -> Forename,
              Target -> Surname,
              Target -> Postcode
              );
            }
          }

        Rows -> EndView( & Scope );
        }
      }
    }
  }

int main( int Count,char *Argument[] )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope( & Heap );
  if ( (Count==2) && (Scope.CreateFile( Argument[1] )) )
    {
    if ( Scope.BeginTransaction() )
      {
      INDEX Index;

      if ( CreateIndex( & Index ) )
        { FindKey( Details[2].Forename,& Index ); }

      Scope.EndTransaction();
      }

    Scope.CloseFile();
    }

  return 0;
  }
```

<u>Example-04-11</u>

We have now reached probably the most complex example of the Rockall-DB library in this manual. The first significant change from 'Example-04-10' is the addition of a the `'NewKey'` function and the introduction of a new class called `'ROW_SET'`.

When we did *<u>not</u>* support duplicate keys every entry in the `'Index'` produced a single `'FILE_ADDRESS'` (i.e. `'Address'`) which pointed directly to the associated `'DETAILS'` structure. However, now we *<u>do</u>* support duplicate keys every entry in the `'Index'` produces a single `'FILE_ADDRESS'` which instead points directly to a `'ROW_SET'`. A `'ROW_SET'` is simply a collection of `'FILE_ADDRESS'` values stored in sorted order. The `'ROW_SET'` class is derived from the `'SET'` class and is defined as `'SET'` of `'FILE_ADDRESS'` values. All the `'NewKey'` function does is to either add the supplied `'FILE_ADDRESS'` (i.e. `'Address'`) to an existing `'ROW_SET'` (if the key value already exists) or create a new `'ROW_SET'` (if it does not already exist).

The next significant change is towards the end of the example is where we call `'FindValue'` to extract the results. When we did *<u>not</u>* support duplicate keys in 'Example-04-10' all we needed to do was to call `'View'` with associated `'FILE_ADDRESS'` (i.e. `'Address'`) to get the result. However, in 'Example-04-11' the call to `'View'` gives us a `'ROW_SET'` and so we need to call the `'View'` function in the `'ROW_SET'` class to give us the `'Set'` of `'FILE_ADDRESS'` values and its `'Size'`. We then need to `'View'` each `'FILE_ADDRESS'` (i.e. `'Address'`) in the `'Set'` to get all the results.

It is worth highlighting a few interesting features of the `'ROW_SET'` and `'SET'` classes at this point. The call to `'Touch'` in 'Example-04-11' asynchronously loads all of the data associated with the `'FILE_ADDRESS'` values into memory from storage (if they are not already present). While this is probably excessive in this example this feature is particularly valuable in cases where a `'ROW_SET'` is large and `'ExclusiveView'`, `'Update'` or `'View'` will be called on most of the related `'FILE_ADDRESS'` values. The `'ROW_SET'` and `'SET'` classes also support the set operations `'Difference'`, `'Intersect'` and `'Join'` to allow set operations to be performed in a way very similar to taditional relational databases.

Now, let's just say that we changed 'Example-04-11' to do multiple calls to `'FindValue'` with different key values to produce multiple `'ROW_SET'` values. If we called `'Intersect'` on all these `'ROW_SET'` values then the resultant `'ROW_SET'` would contain the `'FILE_ADDRESS'` values where ***all*** the these keys were present. If instead we called `'Join'` then the resultant `'ROW_SET'` would contain the `'FILE_ADDRESS'` values where *any* the these keys were present. Finally, if instead we called `'Difference'` then the resultant `'ROW_SET'` would contain the `'FILE_ADDRESS'` values where *only one* of these keys were present.

Just for completeness we shall now take a quick look as the `'QUEUE'` and `'TREE'` classes, even though there is very little new material to cover in these areas. We shall begin by converting 'Example-04-02' to use a `'QUEUE'` in 'Example-04-12' below:

```
#include "AutomaticHeapScope.hpp"
#include "DatabaseHeap.hpp"
#include "FlexibleString.hpp"
#include "Failure.hpp"
#include "Queue.hpp"
#include "RockallTypes.hpp"

typedef FLEXIBLE_STRING<DATABASE_HEAP,CHAR> MESSAGE;

void PushAndPop( void )
  {
  AUTOMATIC_HEAP_SCOPE<DATABASE_HEAP> Scope;
  static const int MaxMessages = 10;
  MESSAGE Messages[ MaxMessages ] =
    {
    "One", "Two", "Three", "Four", "Five",
    "Six", "Seven", "Eight", "Nine", "Ten"
    };

  if ( Scope.BeginTransaction() )
    {
    QUEUE<DATABASE_HEAP,NO_LOCK,MESSAGE> Queue( & Scope );
    MESSAGE Value;
    int Count;

    for
      (
      Count=0;
      (Count < MaxMessages)
        &&
      (Queue.PushValue( & Scope,Messages[ Count ] ));
      Count ++
      );

    if ( Count < MaxMessages)
      { FAILURE( "Unable to push all the values" ); }

    for
      (
      /* void */;
      (Count > 0) && (Queue.PopValue( & Scope,& Value ));
      Count --
      );

    if ( Count > 0 )
```

```
      { FAILURE( "Unable to pop all the values" ); }
    }
  }

int main( int Count,char *Argument[] )
  {
  DATABASE_HEAP Heap;
  AUTOMATIC_HEAP_SCOPE<DATABASE_HEAP> Scope( & Heap );

  if ( (Count == 2) && (Scope.CreateFile( Argument[1] )) )
    {
    PushAndPop();

    Scope.CloseFile();
    }

  return 0;
  }
```

Example-04-12

Now, although ‘Example-04-02’ and ‘Example-04-12’ appear similar in most respects we see that in ‘Example-04-12’ we use a `‘QUEUE’` instead of a `‘STACK’` and store values of type `‘FLEXIBLE_STRING’` instead of `‘int’`. We have modified the example in this way to highlight that with Rockall-DB it is trivial to build systems such as transactional messaging systems (i.e. such as ‘Message Queueing’).

Now, let’s convert ‘Example-04-09’ to use a `‘TREE’` in place of a `‘HASH’` in ‘Example-04-13’ below:

```
#include <stdio.h>

#include "AutomaticHeapScope.hpp"
#include "AutomaticViewScope.hpp"
#include "Example-04-08.hpp"
#include "Failure.hpp"
#include "RockallNew.hpp"
#include "RockallTypes.hpp"
#include "Tree.hpp"

typedef DATABASE_HEAP HEAP_TYPE;
typedef FIXED_STRING<HEAP_TYPE,char,MaxName> NAME;
typedef TREE<HEAP_TYPE,NO_LOCK,NAME,FILE_ADDRESS,16000,16000> INDEX;

static bool CreateIndex( INDEX *Index )
  {
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope;
  int Count;

  for ( Count=0;Count < MaxDetails;Count ++ )
    {
    DETAILS *Current1 = & Details[ Count ];
    NAME Name1 = Current1 -> Forename;
    NAME Name2 = Current1 -> Surname;
    FILE_ADDRESS Address;
    DETAILS *Current2;

    if
        (
        (ROCKALL_NEW<DETAILS>::New( & Scope,& Address,& Current2 ))
```

```
          &&
        (Index -> NewKey( & Scope,Name1,Address ))
          &&
        (Index -> NewKey( & Scope,Name2,Address ))
        )
      { (*Current2) = (*Current1); }
    else
      { FAILURE( "Unable to add key" ); }
    }

  return (Count == MaxDetails);
  }

static void NewKey( char *Name,INDEX *Index )
  {
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope;
  FILE_ADDRESS Address;
  NAME Key = Name;

  if ( Index -> NewKey( & Scope,Key,& Address ) )
    {
    AUTOMATIC_VIEW_SCOPE<HEAP_TYPE> ViewScope( & Scope );
    DETAILS *Target;

    if ( ViewScope.View( Address,((void**) & Target) ) )
      {
      printf
        (
        "The details are: '%s %s' at '%s'\n",
        Target -> Forename,
        Target -> Surname,
        Target -> Postcode
        );
      }
    }
  }

int main( int Count,char *Argument[] )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope( & Heap );

  if ( (Count==2) && (Scope.CreateFile( Argument[1] )) )
    {
    if ( Scope.BeginTransaction() )
      {
      INDEX Index;

      if ( CreateIndex( & Index ) )
        { NewKey( Details[2].Forename,& Index ); }

      Scope.EndTransaction();
      }

    Scope.CloseFile();
    }

  return 0;
  }
```

Example-04-13

Now, in this case there are barely any changes between ‘Example-04-10’ and ‘Example-04-13’ except for the use of:

```
typedef TREE<HEAP_TYPE,NO_LOCK,NAME,FILE_ADDRESS,16000,16000> INDEX;
```

instead of:

```
typedef HASH<HEAP_TYPE,NO_LOCK,NAME,FILE_ADDRESS> INDEX;
```

Now, let’s focus in on the two values of `‘16000’` that are optional template parameters to the `‘TREE’` class and specify the number of `‘STEMS’` in a `‘BRANCH’` and the number of `‘LEAVES’` on a `‘TWIG’` respectively (i.e. the fan out if the `‘TREE’`). These two values may vary between `‘16’` and `‘16384’` and provide the user with significant control over the shape of the resulting `‘TREE’`. The values of `‘16000’` in this case are fairly outragious and would be far better suited to a very large index. These settings would support up to 256,000,000 nodes with just one level of index (i.e. 16,000 x 16,000 = 256,000,000) and up to 4,096,000,000,000 nodes with just two levels of index (i.e. 16,000 x 16,000 x 16,000 = 4,096,000,000,000). Clearly, this is excessive and the minimum values of `‘16’` would seem for more reasonable in this case. If the minimum values of `‘16’` were used the `‘TREE’` would support up to 256 nodes with one level of index (i.e. 16 x 16 = 256) and up to 4,096 nodes with two levels of index (i.e. 16 x 16 x 16 = 4,096).

The key point highlighted by ‘Example-04-13’ is the massive flexibility of the Rockall-DB library. Typically, a developer has no control over the structure of indexes and consequently the values selected by implementors can sometimes be totally unsuitable for particular applications. We saw that with the two optional parameters set to `‘16000’` that up to 256,000,000 nodes could be stored with one level of index. However, setting the two optional parameters to `‘16’` would require six levels of index for the same sized `‘Tree’`. Clearly, traversing all of these extra levels of indexes could potentially have very serious implications for performance.

A further difference between a `‘HASH’` and a `‘TREE’` is that in a `‘TREE’` all the keys are stored in sorted order. Consequently, an additional function called `‘FindValues’` is available to locate either the closest key or a number of keys within a specific range. Typically, a `‘HASH’` is a better choice when a simple direct lookup is required in main memory whereas a `‘TREE’` tends to be better when the closest key or a range of keys are needed or the structure is stored in a transactional database.

In closing, it is hoped that the reader can better understand the previous claims suggesting that Rockall-DB can support many other database models, such as graph databases, hierarchic databases, object oriented databases or relational databases. While there is no explicit support for these database models within Rockall-DB nonetheless, hopefully it can be seen that because a `‘FILE_ADDRESS’` can refer to any object it is essentially akin to a memory address in languages like C++, C# or Java. Consequently, any data structure that can be created in-memory using these languages can also be transactionally created and stored in a Rockall-DB database.

It is common for transactional databases to effectively put artificial limitations on data structures (i.e. there is seldom support for ‘union’). A joy of Rockall-DB is that it effectively removes such limitations. Moreover, when using Rockall-DB it is possible to build almost any type of transactional database structure. Obviously, there is occasionally some extra work associated with building certain types of transactional data structures. Typically, this involves replacing any memory address with a `'FILE_ADDRESS'` or suitable alternative. In the case of relational data structures which support duplicate keys the `'ROW_SET'` class may need to be used, as outlined in ‘Example-04-11’. Nonetheless, the benefit of this modest extra complexity is that Rockall-DB allows a developer to easily build precisely what they want, how they want it and where they want it.

We have now finished our brief tour of the Rockall-DB core and the Rockall-DB library. We shall continue by reviewing of the Rockall-DB core and Rockall-DB library in greater detail in the reference chapters 5 and 6 respectively.

5. The Rockall-DB Core

In this chapter we will go through the core Rockall-DB functionality step by step to highlight all the main features. We will start with the Rockall-DB heaps and basic functions and then move on towards the more complicated features, such as asynchronous I/O, mirroring, regions and transactions.

5.1. The Rockall-DB Heaps and Types

At the centre of Rockall-DB is a collection of heaps and types. The Rockall-DB types are used internally throughout Rockall-DB and approximate to the following standard C/C++ data types:

Rockall-DB Type	***Standard C/C++ Type***
`BOOLEAN`	`bool`
`CHAR`	`char`
`FILE_ADDRESS`	`long long int`
`FILE_SIZE`	`long long int`
`SBIT16`	`short`
`SBIT32`	`int`
`SBIT64`	`long long int`
`VOID`	`void`
`WCHAR`	`wchar_t`

Table-05-01

The Rockall-DB types are used for consistency, managability and simplicity. Any use of these types by developers is entirely optional and completely at their discretion. The vast majority of Rockall-DB has been written in C/C++ in a `‘namespace’` called `‘ROCKALL’` in an effort to minimize any interference with other source code. All of the above Rockall-DB types may optionally be included into other C/C++ modules by using the following header:

```
#include "RockallTypes.hpp"
```

Alternately, individual specific types (or other functionality) can be accessed directly by using the `‘namespace’` prefix as follows:

```
ROCKALL::BOOLEAN
ROCKALL::CHAR
ROCKALL::FILE_ADDRESS
ROCKALL::FILE_SIZE
ROCKALL::SBIT16
ROCKALL::SBIT32
ROCKALL::SBIT64
ROCKALL::WCHAR
```

If there are issues with the Rockall-DB headers this can be mitigated by setting the following compile time flag:

```
#define DISABLE_ROCKALL_GLOBAL_TYPES 1
```

The Rockall-DB heaps are central to all of the Rockall-DB functionality and are as follows:

Heap Name	Header File	Description
`SINGLE_THREADED_HEAP`	`SingleThreadedHeap.hpp`	A single threaded in-memory heap.
`MULTI_THREADED_HEAP`	`MultiThreadedHeap.hpp`	A multi-threaded in-memory heap.

`TRANSACTIONAL_HEAP`	`TransactionalHeap.hpp`	A multi-threaded in-memory transactional heap.
`DATABASE_HEAP`	`DatabaseHeap.hpp`	A multi-threaded in-file transactional heap
`VIRTUAL_HEAP`	`VirtualHeap.hpp`	The virtual base class for all Rockall-DB heaps.

Table-05-02

An instance of a Rockall-DB heap can be created as shown in ‘Example-05-01’ below:

```
#include "SingleThreadedHeap.hpp"

int main( void )
  {
  SINGLE_THREADED_HEAP Heap;

  return 0;
  }
```

Example-05-01

All the Rockall-DB heaps follow the same block structure rules as any other data type in C/C++. Consequently, when a Rockall-DB heap goes out of scope it automatically deletes any allocations related to the heap. If any of the allocations on a heap have been created with the flag `‘Destroy’` set to `‘True’` (see `‘New’` later) then the associated destructors will be executed prior to the allocations being deleted. This is the case for all of Rockall-DB heaps except for ‘`DATABASE_HEAP`’. A ‘`DATABASE_HEAP`’ does not delete any allocations or support the `‘Destroy’` flag but instead aborts any outstanding transactions and closes the related file. Consequently, all of the allocations and data stored in the file should still be present the next time the file is opened by the `‘OpenFile’` function.

Great care must be taken if a Rockall-DB heap is created outside of a functional code block as shown in ‘Example-05-02’ below:

```
#include "SingleThreadedHeap.hpp"

static SINGLE_THREADED_HEAP Heap;

int main( void )
  { return 0; }
```

Example-05-02

It is vital that the constructor for a Rockall-DB heap is called before its first use and its destructor after its last use. Although this may seem obvious, this requirement can pose problems in some situations. In particular, it should also be noted that some versions of Microsoft Windows automatically abort all threads when a call is made to `‘exit()’` or when executing `‘return’` from `‘main()’`. This can cause Rockall-DB to hang as active threads are sometimes aborted while holding critical resources. A solution to this specific issue is to put all of the code into a Dynamic Link Library ‘dll’ and to load and unload it as shown in ‘Example-05-03’ below:

```
#include "windows.h"

typedef int (*MAIN_IN_DLL)( int Count,char *Argument[] );

int main( int Count,char *Argument[] )
  {
```

```
  HINSTANCE Handle = LoadLibrary( ((LPCWSTR) L"User.dll") );

  if ( Handle != NULL )
    {
    MAIN_IN_DLL MainInDll =
      ((MAIN_IN_DLL) GetProcAddress( Handle,"main" ));

    if ( MainInDll != NULL )
      {
      int Status = MainInDll( Count,Argument );

      if ( FreeLibrary( Handle ) )
        { return Status; }
      }
    }

  return 1;
  }
```

Example-05-03

Clearly, any attempt to use a Rockall-DB heap before the execution of its constructor is complete or after the start of the execution of its destructor is not defined (i.e. grave disorder may result).

A technical specification for the constructor of each of the Rockall-DB heaps is as follows:

```
SINGLE_THREADED_HEAP( VOID )
MULTI_THREADED_HEAP( VOID )
TRANSACTIONAL_HEAP( VOID )
```

and

```
DATABASE_HEAP
  (
  SBIT32 NewMaxTime  = DefaultTime,
  SBIT32 NewMinSpace = DefaultMinSpace,
  SBIT32 NewMaxSpace = DefaultMaxSpace
  )
```

A `'SINGLE_THREADED_HEAP'` is not a thread safe heap and so can only be used by one thread at any instant in time. Its simplicity and limited functionality make it the fastest of the Rockall-DB heaps. A good general programming model is to have one (or more) `'SINGLE_THREADED_HEAP'` per thread, to hold private data that is closely associated with the thread. There are a variety of justifications for this suggestion such as:

1. No locks, so faster private memory allocations for the thread.
2. A threads' private memory allocations will be densely packed reducing cache-line activity, false sharing and paging.
3. A thread can call `'DeleteAll'` (see later) to delete all the allocations at suitable points during its execution to reduce the risk of memory leaks.
4. If a thread fails all of its local memory can be safely and easily cleaned up.

A `‘MULTI_THREADED_HEAP’` is a thread safe heap and so can be used by multiple threads at the same time. While it is typical to have a single `‘MULTI_THREADED_HEAP’` per process for globally shared data, this is arbitary and other programming models can easily be used. It was suggested above that a `‘SINGLE_THREADED_HEAP’` per thread was a good general model. If a threads private data is occasionally deleted by other threads then a `‘MULTI_THREADED_HEAP’` per thread could be used instead of a `‘SINGLE_THREADED_HEAP’` per thread (i.e. because a `‘SINGLE_THREADED_HEAP’` could not safely be used in this situation). Although this change would not offer all of the benefits outlined above it would still preserve most of these benefits and would certainly be much more efficient than having a single global heap for all the allocations and shared by all the threads.

A `‘TRANSACTIONAL_HEAP’` is an extension of a `‘MULTI_THREADED_HEAP’` that supports the Rockall-DB transactional functionality in main memory (i.e. not in files). Although the programming model for a `‘TRANSACTIONAL_HEAP’` is a little more complex, it does support rolling back memory allocations when transactions fail. Obviously, there is a significant cost associated with this functionality due to the associated lock manager and change tracking. Nonetheless, if this overhead is acceptable then a `‘TRANSACTIONAL_HEAP’` can greatly significantly simplify a wide variety of software applications and systems.

Finally, a `‘DATABASE_HEAP’` is an extension of a `‘TRANSACTIONAL_HEAP’` that supports the Rockall-DB transactional functionality within files. It is the only Rockall-DB heap that does not clean up its allocations at the end of its scope, but rather aborts any outstanding transactions and closes any related file. If a program containing a `‘DATABASE_HEAP’` crashes an attempt will be made to automatically recover the file the next time the associated file is opened. This recovery process consists of aborting any incomplete transactions and reversing out the associated changes. In short, a `‘DATABASE_HEAP’` is eseentialy follows the same recovery steps as a traditional transactional database.

The technical specification of a `‘DATABASE_HEAP’` is more complex than the other Rockall-DB heaps and is as follows:

```
DATABASE_HEAP
   (
   SBIT32 NewMaxTime  = DefaultTime,
   SBIT32 NewMinSpace = DefaultMinSpace,
   SBIT32 NewMaxSpace = DefaultMaxSpace
   )
```

The `‘NewMaxTime’` parameter specifies the maximum time (in seconds) an unused block from a file will be cached in main memory. The current `‘DefaultTime’` is 5 minutes (i.e. 300 seconds). After this time, it will be removed from main memory and any associated memory released. If block is used again at some later point it will need to be reloaded in to main memory again from stoarge. The `‘NewMinSpace’` parameter sets a lower bound on the percentage of main memory the current process can use for its Rockall-DB cache. The current `‘DefaultMinSpace’` is 5 (i.e. 5%). The `‘NewMaxSpace’` parameter sets an upper bound on the percentage of main memory the current process can use for its Rockall-DB cache. The current `‘DefaultMaxSpace’` is 80 (i.e. 80%).

The memory consumption of Rockall-DB can briefly exceed these limitations if there is a short-term need for memory. Regardless, it is strongly recommended that no single memory allocation on a `'DATABASE_HEAP'` exceed 25% of the size of main memory (i.e. see `'New'` later). Moreover, wild swings in the amount of memory allocated (i.e. say 25% of main memory) can sometimes provoke severe short-term memory pressure resulting in hard page faults and poor performance. Typically, such situations are uncommon but may occur in data hungry applications (i.e. such as an application that transactionally stores multiple full HD movies).

A Rockall-DB heap is a standard C/C++ class and so follows the associated usual rules. All the related member functions can be called in the normal way as outlined in 'Example-05-04' below:

```
#include "SingleThreadedHeap.hpp"
#include "Failure.hpp"

int main( void )
  {
  SINGLE_THREADED_HEAP Heap;
  int *Data;

  if ( ! Heap.New( ((void**) & Data),sizeof(int) ) )
    { FAILURE( "Unable to allocate an int" ); }

  return 0;
  }
```

Example-05-04

Now, let's move on and examine the core Rockall-DB functions in more detail in the following sections.

5.2. The Rockall-DB Core Functions

All of the Rockall-DB core functions work on any of the Rockall-DB heaps. The precise features and requirements of these functions sometimes vary depending on the type of heap. As an example, the `'New'` and `'Delete'` functions do not need to be within a transaction scope when used on a `'SINGLE_THREADED_HEAP'` or `'MULTI_THREADED_HEAP'` but do need to be within a transaction scope when used on a `'TRANSACTIONAL_HEAP'` or `'DATABASE_HEAP'`. A table of the Rockall-DB core functions covered in this section is as follows:

Function Name	Description
`Delete`	Delete an allocation previously created by `'New'` or `'Resize'` (optionally with its destructor).
`DeleteAll`	Delete all the allocations on a heap (optionally executing all the related destructors).
`Details`	Get the details of an allocation previously created by `'New'` or `'Resize'`.
`Feature`	Test the features supported by a heap.
`GetError`	Get any error code relating to the last Rockall-DB call on a heap.
`New`	Create a new allocation on a heap.
`Resize`	Adjust the size of an allocation previously created by `'New'` or `'Resize'`.
`SetError`	Set the error code to a user supplied value.
`Size`	Compute the total size of all the allocations on a heap.
`Walk`	Walk a heap executing a user supplied callback for every allocation on a heap.

Table-05-03

All of the functions listed in ‘Table-05-03’ are described in detail in the following sections in the order they appear the table above.

5.2.1. The Delete Function

The `'Delete'` function frees space previously allocated by a call to the `'New'` or `'Resize'` functions. The `'Delete'` function requires an active transaction when used on a `'DATABASE_HEAP'` or `'TRANSACTIONAL_HEAP'` but not otherwise (see `'BeginTransaction'` for more information about transactions).

A technical specification of the variants of the `'Delete'` function are as follows:

```
VIRTUAL BOOLEAN Delete
  (
  FILE_ADDRESS Address,
  BOOLEAN NoDestroy = False,
  BOOLEAN Zero = False
  )

VIRTUAL BOOLEAN Delete
  (
  VOID *Data,
  BOOLEAN NoDestroy = False,
  BOOLEAN Zero = False
  )
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	Needs a Transaction
MULTI_THREADED_HEAP	Supported
SINGLE_THREADED_HEAP	Supported
TRANSACTIONAL_HEAP	Needs a Transaction

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	Needs a Transaction
MULTI_THREADED_HEAP	Supported
SINGLE_THREADED_HEAP	Supported
TRANSACTIONAL_HEAP	Needs a Transaction

When ‘`Delete`’ is used in conjunction with a ‘`MULTI_THREADED_HEAP`’ or a ‘`SINGLE_THREADED_HEAP`’ the allocation is deleted before the call returns. When ‘`Delete`’ is used with a ‘`DATABASE_HEAP`’ or ‘`TRANSACTIONAL_HEAP`’ the allocation is deleted when the related transaction completes. If the transaction aborts, the allocation will be left unchanged and it will not be deleted. An optimization can occur in the situation where an allocation is created and deleted within the same transaction, in this case a ‘`Delete`’ typically occurs before the call returns.

The `'New'` and `'Resize'` functions provide up to two addresses when creating an a new or modified allocation. The first is a `'FILE_ADDRESS'` (called `'Address')` which relates to the offset of the allocation in the file (in bytes). The second is a `'VOID*'` (called `'Data')` which is a pointer to the allocation in main memory. Either of these addresses may be used in conjunction with corresponding version of `'Delete'` to destroy it. These two addresses contain the same value and so are interchangable giving the same level of performance for all of the Rockall-DB heaps except `'DATABASE_HEAP'.` A `'DATABASE_HEAP'` strongly prefers a `'FILE_ADDRESS'` (i.e. `'Address')` in all situations and will need to compute the `'FILE_ADDRESS'` if it is not supplied in the call.

The `'NoDestroy'` parameter is the complement of the `'Destroy'` parameter in the `'New'` function. When an allocation is created with `'Destroy'` set to `'True'` (see `'New'` for more details) its destructor will be automatically called when it is deleted unless `'NoDestroy'` is set to `'True'` in the associated `'Delete'` call.

The `'Zero'` parameter will overwrite the space used by an allocation after it has been deleted with zeros. This will occur automatically when using a `'DATABASE_HEAP'` or when using the debugging build of Rockall-DB. This is a security feature and offers no other benefits.

The ‘`Delete`’ function will return ‘`True`’ if it was able to find and delete the allocation and ‘`False`’ otherwise. A call to ‘`Delete`’ on any unallocated space in a Rockall-DB heap will not damage the heap or the related area of memory. A call to ‘`Delete`’ on an area of memory unrelated to Rockall-DB should almost never damage it, but it is remotely possible that it will in some rare situations.

When ‘`Delete`’ returns ‘`False`’ more information is usually available by calling the ‘`GetError`’ function to return the error code.

5.2.2. The DeleteAll Function

The ‘`DeleteAll`’ function deletes all of the allocated memory on a Rockall-DB heap. When ‘`DeleteAll`’ is called there must not be any active transactions on the heap.

A technical specification of the ‘`DeleteAll`’ function is as follows:

```
VIRTUAL BOOLEAN DeleteAll
  (
  BOOLEAN NoDestroy = False,
  BOOLEAN Zero = False
  )
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	No Transactions
MULTI_THREADED_HEAP	Supported
SINGLE_THREADED_HEAP	Supported
TRANSACTIONAL_HEAP	No Transactions

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	No Transactions
MULTI_THREADED_HEAP	Supported
SINGLE_THREADED_HEAP	Supported
TRANSACTIONAL_HEAP	No Transactions

The ‘`NoDestroy`’ parameter prevents any destructors from being called for any of the allocations deleted by ‘`DeleteAll`’ (see ‘`Delete`’ above for more details).

The ‘`Zero`’ parameter will overwrite the entire heap after all of the allocations have been deleted with zeros. This will occur automatically when using a ‘`DATABASE_HEAP`’ or when using the debugging build of Rockall-DB. This is a security feature and offers no other benefits.

A call to ‘`DeleteAll`’ on a ‘`DATABASE_HEAP`’ is not transactional but it is atomic. Consequently, everything or nothing on the heap will be deleted. Regardless, in this specific situation it is often both easier and faster to simply close the file and delete it.

The ‘`DeleteAll`’ function will return ‘`True`’ if it was able to delete the heap or ‘`False`’ otherwise.

When ‘`DeleteAll`’ returns ‘`False`’ more information is usually available by calling the ‘`GetError`’ function to return the error code.

5.2.3. The Details Function

The ‘`Details`’ function provides information about an allocation previously returned by the ‘`New`’ or ‘`Resize`’ functions. The ‘`Details`’ function does not require a transaction regardless of the type of Rockall-DB heap.

A technical specification of the variants of the ‘`Details`’ function are as follows:

```
VIRTUAL BOOLEAN Details
  (
  FILE_ADDRESS Address,
  BOOLEAN *Destroy = NULL,
  FILE_SIZE *Space = NULL
  )

VIRTUAL BOOLEAN Details
  (
  VOID *Data,
  BOOLEAN *Destroy = NULL,
  FILE_SIZE *Space = NULL
  )
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	Supported
MULTI_THREADED_HEAP	Supported
SINGLE_THREADED_HEAP	Supported
TRANSACTIONAL_HEAP	Supported

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	Supported
MULTI_THREADED_HEAP	Supported
SINGLE_THREADED_HEAP	Supported
TRANSACTIONAL_HEAP	Supported

The `‘New’` and `‘Resize’` functions provide up to two addresses when creating a new or modified allocation. The first is a `‘FILE_ADDRESS’` (called `‘Address’)` which relates to the offset of the allocation in the file (in bytes). The second is a `‘VOID*’` (called `‘Data’)` which is a pointer to the space in main memory. Either of these addresses may be used in conjunction with corresponding version of `‘Details’` to examine an allocation. These two addresses contain the same value and so are interchangable giving the same level of performance for all of the Rockall-DB heaps except `‘DATABASE_HEAP’`. A `‘DATABASE_HEAP’` strongly prefers a `‘FILE_ADDRESS’` (i.e. `‘Address’)` in all situations and will need to compute the `‘FILE_ADDRESS’` if it is not supplied in the call.

The `‘Destroy’` parameter returns a `‘BOOLEAN’` value indicating the setting of the `‘Destroy’` parameter on the call to `‘New’` when the space was originally allocated. If the value of the `‘Destroy’` parameter is `‘NULL’` (i.e. the default) this parameter it will do nothing.

The `‘Space’` parameter returns the number of bytes occupied by an allocation. The `‘Size’` requested in a call to `‘New’` or `‘Resize’` functions is viewed as a minimum in Rockall-DB and all of the Rockall-DB heaps routinely supply additional space for alignment, fragmentation and performance reasons (see `‘New’` for more details). If the value of the `‘Space’` parameter is `‘NULL’` (i.e. the default) this parameter it will do nothing.

The `‘Details’` function will return `‘True’` if it was able to find the allocation or `‘False’` otherwise. A call to `‘Details’` on an area of memory unrelated to Rockall-DB should almost never return `‘True’`, but it is remotely possible in some rare situations.

When `‘Details’` returns `‘False’` more information is usually available by calling the `‘GetError’` function to return the error code.

5.2.4. The Feature Function

The `‘Feature’` function provides information about features supported by a specific Rockall-DB heap. The `‘Feature’` function does not require a transaction regardless of the type of Rockall-DB heap.

A technical specification of the `‘Feature’` function is as follows:

```
BOOLEAN Feature
  (
  SBIT32 Features
  )
```

Rockall-DB Heap	Comments
DATABASE_HEAP	Supported
MULTI_THREADED_HEAP	Supported
SINGLE_THREADED_HEAP	Supported
TRANSACTIONAL_HEAP	Supported

AUTOMATIC_HEAP_SCOPE	Comments
DATABASE_HEAP	Supported
MULTI_THREADED_HEAP	Supported
SINGLE_THREADED_HEAP	Supported
TRANSACTIONAL_HEAP	Supported

The '`Features`' parameter should contain one or more of the feature flags joined together (i.e. using the arithmetic 'or' operator '|' to join the flags). The feature flags are specified in the header '`#include "RockallTypes.hpp"`'. The currently supported feature flags are as follows:

Feature Flag	***Set for the following Rockall-DB Heaps***
`RockallDatabaseHeap`	`DATABASE_HEAP`
`RockallMultiThreadedHeap`	`MULTI_THREADED_HEAP`
`RockallSingleThreadedHeap`	`SINGLE_THREADED_HEAP`
`RockallTransactionalHeap`	`TRANSACTIONAL_HEAP`
`RockallFileSupport`	`DATABASE_HEAP`

`RockallLockSupport`	`DATABASE_HEAP,`
	`MULTI_THREADED_HEAP,`
	`TRANSACTIONAL_HEAP`

`RockallMemorySupport`	`DATABASE_HEAP,`
	`MULTI_THREADED_HEAP,`
	`SINGLE_THREADED_HEAP,`
	`TRANSACTIONAL_HEAP`

`RockallTransactionalSupport`	`DATABASE_HEAP,`
	`TRANSACTIONAL_HEAP`

The '`Feature`' function has been coded to return a static '`BOOLEAN`' value in most cases. This is to help optimize the code generated by the C++ compiler as outlined in 'Example-05-05' below:

```
#include "SingleThreadedHeap.hpp"
#include "Failure.hpp"
#include "RockallTypes.hpp"

int main( void )
  {
  SINGLE_THREADED_HEAP Heap;
  int *Data;

  if ( Heap.Feature( RockallTransactionalSupport ) )
    {
    if
        (
        (! Heap.BeginTransaction())
          ||
        (! Heap.New( ((void**) & Data),sizeof(int) ))
          ||
        (! Heap.EndTransaction())
        )
      { FAILURE( "Unable to allocate an int" ); }
    }
  else
```

```
    {
    if ( ! Heap.New( ((void**) & Data),sizeof(int) ) )
      { FAILURE( "Unable to allocate an int" ); }
    }

  return 0;
  }
```

Example-05-05

The call to `'Heap.Feature( RockallTransactionalSupport )'` in this case will produce the constant value `'False'` at compile time, as a `'SINGLE_THREADED_HEAP'` does not support transactions. Consequently, the C++ compiler can now optimize away the `'if'` statement as the condition is known to always be `'False'`. So, it will be 'as-if' the code had been written as shown in 'Example-05-06' below:

```
#include "SingleThreadedHeap.hpp"
#include "Failure.hpp"
#include "RockallTypes.hpp"

int main( void )
  {
  SINGLE_THREADED_HEAP Heap;
  int *Data;

  if ( ! Heap.New( ((void**) & Data),sizeof(int) ) )
    { FAILURE( "Unable to allocate an int" ); }

  return 0;
  }
```

Example-05-06

A large part of the Rockall-DB library has been written to make use of this feature to automatically optimize away pieces of code that do not apply to particular Rockall-DB heaps. Specifically, this has been done in connection with locks, transactions and unsupported functions (see the `'AUTOMATIC_HEAP_SCOPE'` and `'NO_LOCK'` classes in `'AUTOMATIC_HEAP_SCOPE.hpp'` and `'NO_LOCK.hpp'` respectively).

5.2.5. The GetError Function

The `'GetError'` function provides an error code when a call to a Rockall-DB function returns `'False'`. The `'GetError'` function does not require a transaction regardless of the type of Rockall-DB heap.

A technical specification of the `'GetError'` function is as follows:

```
VIRTUAL BOOLEAN GetError
  (
  SBIT32 *ErrorNumber
  )
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	Supported
MULTI_THREADED_HEAP	Supported
SINGLE_THREADED_HEAP	Supported
TRANSACTIONAL_HEAP	Supported

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	Supported
MULTI_THREADED_HEAP	Supported
SINGLE_THREADED_HEAP	Supported
TRANSACTIONAL_HEAP	Supported

The `'ErrorNumber'` parameter will contain any available error code if a call to `'GetError'` returns `'True'`. The error codes are specified in the header '`#include "Error.hpp"`'. A sample of the currently specified error codes are as follows:

Error Name	***Value***	***Description***
`NoError`	`0`	There is no error code.
`ErrorNoActiveHeap`	`1000`	There is no active heap available for use.
`ErrorUnsupported`	`1001`	The call is not supported in this context.
`ErrorNoAvailableMemory`	`1002`	There is not enough available free memory space.
`ErrorIncorrectParameter`	`1003`	A parameter to the call is invalid.
`ErrorSettingSystemValue`	`2000`	An internal error occurred.
`ErrorLockingRootPage`	`2001`	An internal error occurred.
`ErrorNoSpaceForRootPage`	`2002`	An internal error occurred.
`ErrorCreatingFile`	`2003`	There was an error while trying to create a new file. Typically, the file already exists.
`ErrorFileAlreadyActive`	`2004`	An internal error occurred.
`ErrorOpeningFile`	`2005`	There was an error while trying to open an existing file. Typically, the file does not exist, is protected or locked by another user.
`ErrorReadingSystemValue`	`2006`	An internal error occurred.
`ErrorNoCreateOrOpenFile`	`2007`	There has been no successful call to 'CreateFile()' or 'OpenFile()' prior the current call.
`ErrorActiveTransaction`	`2008`	There is a existing active transaction.
`ErrorNoActiveTransaction`	`2009`	There is no active transaction available for use.
`ErrorNoActiveLock`	`2010`	There is no active lock to use.
`ErrorDeadlock`	`3000`	The request appeared to lead to a deadlock. Release the necessary resources and try again.

5.2.6. The New Function

The `'New'` function allocates space on a Rockall-DB heap. The `'New'` function requires an active transaction when used with a `'DATABASE_HEAP'` or `'TRANSACTIONAL_HEAP'` but not otherwise (see `'BeginTransaction'` for more information about transactions).

A technical specification of the variants of the `'New'` function are as follows:

```
VIRTUAL BOOLEAN New
  (
  FILE_ADDRESS *Address,
  VOID **Data,
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	Needs a Transaction
MULTI_THREADED_HEAP	Supported
SINGLE_THREADED_HEAP	Supported
TRANSACTIONAL_HEAP	Needs a Transaction

AUTOMATIC_HEAP_SCOPE	***Comments***

DATABASE_HEAP	Needs a Transaction
MULTI_THREADED_HEAP	Supported
SINGLE_THREADED_HEAP	Supported
TRANSACTIONAL_HEAP	Needs a Transaction

```
  FILE_SIZE Size,
  FILE_SIZE *Space = NULL,
  BOOLEAN Destroy = False,
  BOOLEAN Zero = False
  )

VIRTUAL BOOLEAN New
  (
  VOID **Data,
  FILE_SIZE Size,
  FILE_SIZE *Space = NULL,
  BOOLEAN Destroy = False,
  BOOLEAN Zero = False
  )
```

The `'New'` function allocates space in main memory for all Rockall-DB heaps except a `'DATABASE_HEAP'` where it will allocate space in the associated file and in main memory at the same time. The only mandatory parameters for `'New'` are `'Data'` and `'Size'`. All the other parameters may be omitted if desired.

The `'Address'` parameter will contain the offset of the allocation in the file (in bytes) for a `'DATABASE_HEAP'` but for all other heaps the value will match the value of the `'Data'` parameter when a call to `'New'` returns `'True'`. The `'Delete'`, `'Details'` and `'Resize'` functions will accept either the `'Address'` or `'Data'` parameter for all operations. However, a `'DATABASE_HEAP'` strongly prefers a `'FILE_ADDRESS'` (i.e. `'Address'`) in all situations and will need to compute a `'FILE_ADDRESS'` if it is not supplied in the call.

The `'Data'` parameter will contain a pointer to the allocation in main memory when a call to `'New'` returns `'True'`. If the allocation `'Size'` is a power of two (and less than or equal to 4K) then the memory allocated will always fall on the natural power of two alignment boundary. Moreover, Rockall-DB expends quite a bit of effort to align allocations, compact free space and minimize fragmentation. This effort typically pays off in long running applications where the impact of poor alignment, low data density and fragmentation are most keenly felt.

All allocations in Rockall-DB are stored end-to-end with ***no*** heap data between them except at the beginning and end of pages. This structure enhances the density of the memory allocations and so aids performance. However, it also means that any memory overrun will likely to damage user data rather than heap data. A collection of special ‘debug’ builds are available in Rockall-DB to help find such problems. Additionally, the `'Walk'` function (see later) can be used to traverse all the nearby allocations.

The `'Size'` parameter should contain the number of bytes required for a new allocation. However, in Rockall-DB this is viewed as a minimum and often some extra bytes will be allocated for alignment, fragmentation and performance reasons. These extra bytes are available for general use and the actual size of the allocation can be found via the `'Space'` parameter. It is strongly advised that the ‘`Size`’ parameter should never be greater than 25% of the available DRAM when using a `'DATABASE_HEAP'`. Otherwise, a grave short term need for memory may be experienced leading to multiple hard page faults.

The `‘Space’` parameter will contain the actual number of bytes allocated when a call to `‘New’` returns ‘True’. All of these bytes are available for use in the usual way. If the value of the `‘Space’` parameter is `‘NULL’` (i.e. the default) this parameter it will do nothing.

The `‘Destroy’` parameter can be set to `‘True’` or `‘False’` (i.e. the default). If set to `‘True’` then the class being allocated must be inherited for the `‘VIRTUAL_DESTRUCTOR’` class (see `‘VirtualDestructor.hpp’`) to ensure there is a v-table related to the object and that the first entry points to the objects destructor. A fuller discussion of this topic is presented in chapter 3 starting with ‘Example-03-04’ and ending with ‘Example-03-13’. When the `‘Destroy’` parameter is set to `‘True’` Rockall-DB will automatically call an objects destructor when it is deleted by any call to `‘Delete’` or `‘DeleteAll’` for all the Rockall-DB heaps except for a `‘DATABASE_HEAP’`. If a call to `‘Resize’` is made in connection with the allocation the property will continue persist and can only be disabled by setting `‘NoDestroy’` to `‘True’` on the associated call to `‘Delete’` or `‘DeleteAll’`. A `‘DATABASE_HEAP’` will return `‘False’` on any call to `‘New’` with the `‘Destroy’` parameter is set to `‘True’`. This is necessary as `‘Destroy’` requires a v-table and such structures cannot be reliably stored and used in a transactional database.

The `‘Zero’` parameter can be set to `‘True’` or `‘False’` (i.e. the default). If set to `‘True’` the contents of any new allocation will be zeroed. If set to `‘False’` the contents of the any new allocation are undefined.

It should be noted that memory addresses, function pointers, operating system handles and similar structures should not be stored in a `‘DATABASE_HEAP’`. There is nothing in Rockall-DB to prevent this and it will not affect Rockall-DB in any way. However, the contents of a `‘DATABASE_HEAP’` typically last well beyond the execution of a single program and so these values are almost guaranteed to be invalid if used at some later date.

When `‘New’` returns `‘False’` more information is usually available by calling the `‘GetError’` function to return the error code.

5.2.7. The Resize Function

The `‘Resize’` function resizes an allocation on a Rockall-DB heap. The `‘Resize’` function requires an active transaction when used with a `‘DATABASE_HEAP’` or `‘TRANSACTIONAL_HEAP’` but not otherwise (see `‘BeginTransaction’` for more information about transactions).

A technical specification of the variants of the `‘Resize’` function are as follows:

```
VIRTUAL BOOLEAN Resize
   (
   FILE_ADDRESS *Address,
   VOID **Data,
   FILE_SIZE NewSize,
   FILE_SIZE *Space = NULL,
   BOOLEAN Zero = False
   )
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	Needs a Transaction
MULTI_THREADED_HEAP	Supported
SINGLE_THREADED_HEAP	Supported
TRANSACTIONAL_HEAP	Needs a Transaction

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	Needs a Transaction
MULTI_THREADED_HEAP	Supported
SINGLE_THREADED_HEAP	Supported
TRANSACTIONAL_HEAP	Needs a Transaction

```
VIRTUAL BOOLEAN Resize
  (
  VOID **Data,
  FILE_SIZE NewSize,
  FILE_SIZE *Space = NULL,
  BOOLEAN Zero = False
  )
```

The `'Resize'` function resizes an allocation in main memory for all Rockall-DB heaps except a `'DATABASE_HEAP'`, where it resizes the allocation in the associated file and in main memory. The only mandatory parameters for `'Resize'` are `'Data'` and `'NewSize'`. All the other parameters may be omitted if desired.

A call to `'Resize'` is functionally the same as calling `'New'` to create a new allocation, copying the contents of the existing allocation into the new space and finally deleting the existing allocation. A call to `'Resize'` will *always move an allocation* unless the `'Space'` that needs to be allocated is unchanged.

The '`Resize`' function does not call any constructors or destructors but rather uses a simple copy to move data between allocations. When '`Resize`' is used with a '`MULTI_THREADED_HEAP`' or a '`SINGLE_THREADED_HEAP`' the existing allocation is deleted before the call returns. When '`Resize`' is used with a '`DATABASE_HEAP`' or '`TRANSACTIONAL_HEAP`' the existing allocation is only deleted when the transaction completes. If the transaction aborts, the existing allocation will be left unchanged and instead the new allocation will be deleted. An optimization may occur in the situation where an allocation is created and resized within the same transaction, in this case the deletion typically occurs before before the call returns.

The `'Address'`, `'Data'` and `'Space'` parameters operate in the same way as described above in the `'New'` function above and so are outlined again here.

The `'NewSize'` parameter operates in the same way as `'Size'` parameter described above in the `'New'` function above and so is not outlined again here.

The `'Zero'` parameter can be set to `'True'` or `'False'` (i.e. the default). If set to `'True'` the contents of any *extra* space in the allocation will be zeroed. If set to `'False'` the contents of any *extra* space in the allocation will be undefined.

When `'Resize'` returns `'False'` more information is usually available by calling the `'GetError'` function to return the error code.

5.2.5. The SetError Function

The `'SetError'` function allows the Rockall-DB error code to be set to any user supplied value and always returns `'False'`. The `'SetError'` function does not require a transaction regardless of the type of Rockall-DB heap.

A technical specification of the `'SetError'` function is as follows:

Rockall-DB Heap	*Comments*
DATABASE_HEAP	Supported
MULTI_THREADED_HEAP	Supported
SINGLE_THREADED_HEAP	Supported

```
VIRTUAL BOOLEAN SetError
  (
  SBIT32 ErrorNumber
  )
```

TRANSACTIONAL_HEAP	Supported

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	Supported
MULTI_THREADED_HEAP	Supported
SINGLE_THREADED_HEAP	Supported
TRANSACTIONAL_HEAP	Supported

The `‘ErrorNumber’` parameter may contain and value and this value will typically be returned by any later call to `‘GetError’` that returns `‘True’`.

5.2.8. The Size Function

The `‘Size’` function computes the size of all the allocations on a Rockall-DB Heap. The `‘Size’` function does not require a transaction regardless of the type of Rockall-DB heap.

A technical specification of the `‘Size’` function is as follows:

```
VIRTUAL BOOLEAN Size
  (
  FILE_SIZE *Space
  )
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	Supported
MULTI_THREADED_HEAP	Supported
SINGLE_THREADED_HEAP	Supported
TRANSACTIONAL_HEAP	Supported

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	Supported
MULTI_THREADED_HEAP	Supported
SINGLE_THREADED_HEAP	Supported
TRANSACTIONAL_HEAP	Supported

The `‘Size’` function computes the size of a Rockall-DB heap using the `‘Walk’` function outlined below. The calculation of the `‘Size’` of a heap involves visiting every page and so consumes a significant number of CPU cycles.

The `‘Space’` parameter will contain the size of the heap (in bytes) if a call to `‘Size’` returns ‘True’.

When `‘Size’` returns `‘False’` more information is usually available by calling the `‘GetError’` function to return the error code.

5.2.9. The Walk Function

The `‘Walk’` function calls a user supplied function for of all the allocations in a Rockall-DB heap. When `‘Walk’` is called there must *not* be any active transactions on the heap.

A technical specification of the `‘Walk’` function is as follows:

```
VIRTUAL BOOLEAN Walk
  (
  HEAP_WALK_FUNCTION WalkFunction,
  VOID *UserValue = NULL
  )

typedef BOOLEAN (*HEAP_WALK_FUNCTION)
  (
  BOOLEAN         Active,
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	No Transactions
MULTI_THREADED_HEAP	Supported
SINGLE_THREADED_HEAP	Supported
TRANSACTIONAL_HEAP	No Transactions

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	No Transactions
MULTI_THREADED_HEAP	Supported
SINGLE_THREADED_HEAP	Supported
TRANSACTIONAL_HEAP	No Transactions

```
  FILE_ADDRESS    Address,
  VOID            *Data,
  BOOLEAN         Destroy,
  FILE_SIZE       Space,
  VOID            *UserValue
  )
```

The `'Walk'` function calls the `'WalkFunction'` for every allocation in a Rockall-DB heap. The parameters passed to each invocation of the `'WalkFunction'` are outlined below:

The `'Active'` parameter is set to `'True'` if the memory allocation is in use and `'False'` if it free.

The `'Address'` parameter is the offset of the current allocation in the file (in bytes) for a `'DATABASE_HEAP'` or the same value as `'Data'` for all other types of Rockall-DB heap.

The `'Data'` parameter is a pointer to the allocation in memory and is suitable for ***reading*** its contents.

The `'Destroy'` parameter is `'True'` if `'Destroy'` was set to `'True'` on the call to `'New'` when the allocation was created, otherwise it will be `'False'`.

The `'Space'` parameter is contains the number of bytes in the allocation. This may be larger than the `'Size'` requested in the call to `'New'` or `'Resize'`.

The `'UserValue'` parameter contains the `'UserValue'` initially passed on the call to `'Walk'`. This enables a user supplied value (or a pointer to a defined user structure) to be passed to all invocations of the `'WalkFunction'`.

If the `'WalkFunction'` returns `'False'` then the call to `'Walk'` will terminate immediately and will return `'False'`.

As mentioned above the `'Size'` function is implemented using the `'Walk'` function and an outline of its implementation is shown in 'Example-05-07' below:

```
#include "SingleThreadedHeap.hpp"
#include "Failure.hpp"
#include "RockallTypes.hpp"

BOOLEAN WalkFunction
    (
    BOOLEAN         Active,
    FILE_ADDRESS    Address,
    VOID            *Data,
    BOOLEAN         Destroy,
    FILE_SIZE       Space,
    VOID            *UserValue
    )
  {
  if ( Active )
    {
    FILE_SIZE *Size = ((FILE_SIZE*) UserValue);
    (*Size) += Space;
    }

  return True;
  }

int main( void )
  {
  SINGLE_THREADED_HEAP Heap;
  FILE_SIZE Size = 0;

  if ( ! Heap.Walk( WalkFunction,& Size ) )
    { FAILURE( "Unable to walk heap" ); }

  return 0;
  }
```

Example-05-07

When ‘`Walk`’ returns ‘`False`’ more information is often available by calling the ‘`GetError`’ function to return the error code.

5.3. The Rockall-DB Transactional Functions

The Rockall-DB transactional functions are available on the Rockall-DB transactional heaps which are ‘`DATABASE_HEAP`’ and ‘`TRANSACTIONAL_HEAP`’ (see section 5.1 for a summary of the Rockall-DB heaps). A table of all the Rockall-DB functions covered in this section is as follows:

Function Name	Description
`CreateFile`	Create and activate a ‘`DATABASE_HEAP`’ or ‘`TRANSACTIONAL_HEAP`’.
`CreateRegion`	Create a new separate region within a heap.
`BeginTransaction`	Create a new transaction scope for the current thread on a heap.
`ChangeRegion`	Change the current active region in a heap.
`ExclusiveView`	Exclusively view an allocation previously created by ‘`New`’ or ‘`Resize`’.
`EndExclusiveView`	Release all the resources previously claimed by a call to ‘`ExclusiveView`’.
`JoinTransaction`	Join the current thread to an existing transaction (i.e. parallel processing).
`LeaveTransaction`	Leave an existing transaction (i.e. the converse of ‘`JoinTransaction`’).
`Update`	Exclusively update an allocation previously created by ‘`New`’ or ‘`Resize`’.
`View`	Non-exclusively view an allocation previously created by ‘`New`’ or ‘`Resize`’.
`EndView`	Release all the resources previously claimed by a call to ‘`View`’.
`EndTransaction`	End a transaction, commit any updates and release any related resources.
`DeleteRegion`	Delete a separate region within a heap along with all of its related allocations.
`CloseFile`	Close and deactivate a ‘`DATABASE_HEAP`’ or ‘`TRANSACTIONAL_HEAP`’.

Table-05-04

All of the functions listed in ‘Table-05-04’ are described in detail in the following sections in the order they appear the table above.

5.3.1. The CreateFile Function

The ‘`CreateFile`’ function prepares a ‘`DATABASE_HEAP`’ or ‘`TRANSACTIONAL_HEAP`’ for use. When ‘`CreateFile`’ is called there must not be any active transactions on the heap.

A technical specification of the variants of the ‘`CreateFile`’ function are as follows:

Rockall-DB Heap	***Comments***
DATABASE_HEAP	No Transaction

MULTI_THREADED_HEAP	Not Supported
SINGLE_THREADED_HEAP	Not Supported
TRANSACTIONAL_HEAP	No Transaction

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	No Transaction
MULTI_THREADED_HEAP	Does nothing
SINGLE_THREADED_HEAP	Does nothing
TRANSACTIONAL_HEAP	No Transaction

```
VIRTUAL BOOLEAN CreateFile
  (
  CHAR *FileName,
  ENCRYPTION_FUNCTION Function = NULL,
  VOID *UserValue = NULL
  )

VIRTUAL BOOLEAN CreateFile
  (
  WCHAR *FileName,
  ENCRYPTION_FUNCTION Function = NULL,
  VOID *UserValue = NULL
  )

typedef VOID (*ENCRYPTION_FUNCTION)
  (
  FILE_ADDRESS    Address,
  VOID            *Data,
  BOOLEAN         Decrypt,
  BOOLEAN         Encrypt,
  FILE_SIZE       Size,
  VOID            *UserValue
  )
```

The ‘`CreateFile`’ function comes in both ‘ACSII’ and ‘Unicode’ styles. All of the parameters are optional for a ‘`TRANSACTIONAL_HEAP`’ and have no functional value (i.e. they do nothing). Alternately, all the parameters have a meaning in regards to a ‘`DATABASE_HEAP`’ and the ‘`FileName`’ parameter is mandatory.

The ‘`FileName`’ parameter is the name of the new file to be created by Rockall-DB. The new file will automatically be formatted as a Rockall-DB database and must *not* already exist.

The optional ‘`Function`’ parameter allows a user supplied ‘`ENCRYPTION_FUNCTION`’ to be supplied which will be called by Rockall-DB every time a data block is read or written to the file. When a block is read the ‘`ENCRYPTION_FUNCTION`’ will be called just after the read completes and it is required to immediately decrypt it. When a block is written the ‘`ENCRYPTION_FUNCTION`’ will be called just before the write starts and the it is required to immediately encrypt it. The meaning of the parameters supplied to the ‘`ENCRYPTION_FUNCTION`’ is as follows:

The ‘`Address`’ parameter is the offset in the file (in bytes) of the start of block.

The ‘`Data`’ parameter is the memory address of the data to be encrypted or decrypted.

The ‘`Decrypt`’ parameter will be ‘`True`’ if the ‘`Data`’ needs to be decrypted just after it has been read otherwise it will be ‘`False`’.

The ‘`Encrypt`’ parameter will be ‘`True`’ if the ‘`Data`’ needs to be encrypted just before it is written otherwise it will be ‘`False`’.

The ‘`Size`’ parameter contains the number of bytes in the area to be encrypted or decrypted.

The `‘UserValue’` parameter contains the `‘UserValue’` parameter initially passed to `’CreateFile’`. This enables a user supplied value (or a pointer to a defined user structure) to be passed to all invocations of the ‘`ENCRYPTION_FUNCTION`’.

Technically, the ‘`ENCRYPTION_FUNCTION`’ does not actually need to decrypt and encrypt the data. Instead, it could use the opportunity to do some other form of processing on the data. Regardless, damaging the data in any way can lead to grave disorder in Rockall-DB and will most likely corrupt the database.

When `‘CreateFile’` returns `‘False’` more information is usually available by calling the `‘GetError’` function to return the error code.

5.3.2. The CreateRegion Function

The `‘CreateRegion’` function is intended for advanced users and in effect creates multiple sub-heaps within a single `‘DATABASE_HEAP’` or `‘TRANSACTIONAL_HEAP’`. These sub-heaps can then be used to keep closely related data together which typically improves data density and performance. When `‘CreateRegion’` is called there must not be any active transactions on the heap.

A technical specification for `‘CreateRegion’` is as follows:

```
VIRTUAL BOOLEAN CreateRegion
  (
  SBIT32 *NewRegionID
  )
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	No Transaction
MULTI_THREADED_HEAP	Not Supported
SINGLE_THREADED_HEAP	Not Supported
TRANSACTIONAL_HEAP	No Transaction

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	No Transaction
MULTI_THREADED_HEAP	Does nothing
SINGLE_THREADED_HEAP	Does nothing
TRANSACTIONAL_HEAP	No Transaction

We will begin by considering a large multi-terabyte database with two large tables and two large indexes. We could use the `‘CreateRegion’` function to create four separate regions (i.e. one for each table and one for each index). Now, by putting each table and each index in its own private region all the related data will be naturally seperated into four zones within the file. Over time this will significantly enhance the data density and reduce fragmentation, as each region will keep the related data together (i.e. like the chapters in a book). If we did not use the `‘CreateRegion’` function then all the data would be randomly mixed together throughout the file. Over time this significantly reduces the data density and increases fragmentation leading to lower performance.

Typically, regions are not required for smaller databases (i.e. under 100 GB). However, as the size of a database grows the number of file transfers tend to grow due to fragmentation unless steps are taken to keep the all the active data together. The `‘CreateRegion’` function makes this possible by allowing multiple regions to be created within a single `‘DATABASE_HEAP’` or `‘TRANSACTIONAL_HEAP’` to help control the data density, data placement and the associated fragmentation.

The `‘CreateRegion’` function is part of a family of related functions. These are `‘CreateRegion’`, `’ChangeRegion’` and `‘DeleteRegion’`. There is no limit to the number of regions that can be created within a `‘DATABASE_HEAP’` or

`‘TRANSACTIONAL_HEAP’`. A region does nothing until a call is made to `’ChangeRegion’`. At this point, all future calls refer to the new region. Consequently, all subsequent calls to `‘New’` and `‘Delete’` will create and delete allocations in the new region. A call to `‘DeleteAll’` would delete all the allocations in a specific region but leave allocations in other regions untouched. A region is very much like a separate sub-heap within a heap.

The `‘NewRegionID’` parameter contains the `‘ID’` of the new region if `‘CreateRegion’` returns `‘True’`. The `‘NewRegionID’` is required in subsequent calls to `‘ChangeRegion’` and `‘DeleteRegion’`.

A call to `‘CreateRegion’` is not transactional but is atomic and so will not leave any associated `‘DATABASE_HEAP’` or `‘TRANSACTIONAL_HEAP’` in an intermediate state.

When `‘CreateRegion’` returns `‘False’` more information is usually available by calling the `‘GetError’` function to return the error code.

5.3.3. The BeginTransaction Function

The `‘BeginTransaction’` function creates a new transaction for the current thread for a `‘DATABASE_HEAP’` or `‘TRANSACTIONAL_HEAP’`. There may *only be one open transaction per thread* at any time (see `‘AutomaticHeapScope’` for support of nested transactions).

A technical specification of the `‘BeginTransaction’` function is as follows:

```
VIRTUAL BOOLEAN BeginTransaction(
  (
  FILE_ADDRESS *NewTransactionID = NULL,
  TRANSACTION_LOCK<HEAP> *Lock = NULL
  )
```

Rockall-DB Heap	*Comments*
DATABASE_HEAP	New Transaction
MULTI_THREADED_HEAP	Not Supported
SINGLE_THREADED_HEAP	Not Supported
TRANSACTIONAL_HEAP	New Transaction

AUTOMATIC_HEAP_SCOPE	*Comments*
DATABASE_HEAP	New Transaction
MULTI_THREADED_HEAP	Does nothing
SINGLE_THREADED_HEAP	Does nothing
TRANSACTIONAL_HEAP	New Transaction

The `‘BeginTransaction’` function creates a new transaction for the current thread and is a requirement to create a transaction before making any calls to `‘Delete’`, `‘ExclusiveView’`, `‘New’`, `‘Resize’`, `‘SetUserValue’`, `‘Update’` or `‘View’` on a `‘DATABASE_HEAP’` or `‘TRANSACTIONAL_HEAP’`.

All calls to `‘Delete’`, `‘ExclusiveView’`, `‘New’`, `‘Resize’`, `‘SetUserValue’`, `‘Update’` or `‘View’` on a `‘DATABASE_HEAP’` or `‘TRANSACTIONAL_HEAP’` will be undone unless a successful matching call is made to `‘EndTransaction’`. All the above functions also take locks to ensure that other threads using the same memory allocations do so in a consistent manor. The `‘ExclusiveView’`, `‘New’`, `‘Resize’`, `‘SetUserValue’` and `‘Update’` functions claim exclusive locks on the related allocations whereas the `‘View’` function only claims a sharable lock. The `‘Delete’` function marks an allocation for deletion at the end of the transaction but does *not* lock it. Consequently, care must be taken not to do things like calling `‘Update’` on an allocation when a call is made to `‘Delete’` has already been made, as any changes will be lost at the end of the transaction.

A transaction only applies to the specific instance of a heap on which the call to `‘BeginTransaction’` was made. Nonetheless, it does apply to all the regions within this instance of the heap. Consequently, any calls to `‘ChangeRegion’` or subsequent calls to make chnages are still viewed as part of the same transaction.

The optional `‘NewTransactionID’` parameter contains the `‘ID’` of the new transaction if `‘BeginTransaction’` returns `‘True’`. The `‘NewTransactionID’` is required when calling `‘JoinTransaction’` which allows multiple threads to work together on a single transaction (i.e. parallel transactions).

The optional `‘Lock’` parameter will update any supplied `‘TRANSACTION_LOCK<HEAP>’` and make it refer to an internal lock associated with the transaction. The `‘Lock’` may then be used in the usual way to help manage the threads working on the transaction.

The `‘AutomaticHeapScope’` class in the Rockall-DB library can be used to automatically call `‘EndTransaction’` at the end of a scope to make programming more straight-forward and reliable (see the `‘AutomaticHeapScope’` class).

When `‘BeginTransaction’` returns `‘False’` more information is usually available by calling the `‘GetError’` function to return the error code.

5.3.4. The ChangeRegion Function

The `‘ChangeRegion’` function changes the current region on a `‘DATABASE_HEAP’` or `‘TRANSACTIONAL_HEAP’`. The `‘ChangeRegion’` function does not require a transaction regardless of the type of Rockall-DB heap.

A technical specification of the `‘ChangeRegion’` function is as follows:

```
VIRTUAL BOOLEAN ChangeRegion
  (
  SBIT32 NewRegionID,
  SBIT32 *OldRegionID = NULL
  )
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	Supported
MULTI_THREADED_HEAP	Not Supported
SINGLE_THREADED_HEAP	Not Supported
TRANSACTIONAL_HEAP	Supported

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	Supported
MULTI_THREADED_HEAP	Does nothing
SINGLE_THREADED_HEAP	Does nothing
TRANSACTIONAL_HEAP	Supported

A region can be thought of as a sub-heap within a heap. It is intended for advanced users and enables very large data structures to be kept together in main memory or within a file. A new region is created by calling `‘CreateRegion’` and deleted by calling `‘DeleteRegion’`.

The `‘NewRegionID’` parameter must contain a `‘NewRegionID’` previously returned by a successful call to `‘CreateRegion’`.

The optional `‘OldRegionID’` parameter will contain the `‘ID’` of the region prior to the call to `‘ChangeRegion’` if it returns `‘True’`. The `‘OldRegionID’` value can be used in subsequent calls to `‘ChangeRegion’` to set the region back to its original value (see the `‘AutomaticHeapScope’` class for the automatic management of regions).

When a call to `'ChangeRegion'` returns `'True'` all subsequent calls to `'Delete'`, `'DeleteAll'`, `'Details'`, `'ExclusiveView'`, `'New'`, `'Resize'`, `'Size'`, `'Update'`, `'View'` and `'Walk'` will subsequently refer to the new region. Consequently, any calls to these functions for an allocation created in the previous region will return `'False'` as the previous region will no longer be visible.

When `'ChangeRegion'` returns `'False'` more information is usually available by calling the `'GetError'` function to return the error code.

5.3.5. The ExclusiveView Function

The `'ExclusiveView'` function provides an exclusive read-only view of an allocation on a `'DATABASE_HEAP'` or `'TRANSACTIONAL_HEAP'`. The `'ExclusiveView'` function requires an active transaction (see `'BeginTransaction'` for more information about transactions).

A technical specification of the `'ExclusiveView'` function is as follows:

```
VIRTUAL BOOLEAN ExclusiveView
   (
   FILE_ADDRESS Address,
   VOID **Data,
   FILE_SIZE *Space = NULL
   )
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	Needs a Transaction
MULTI_THREADED_HEAP	Not Supported
SINGLE_THREADED_HEAP	Not Supported
TRANSACTIONAL_HEAP	Needs a Transaction

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	Needs a Transaction
MULTI_THREADED_HEAP	Does nothing
SINGLE_THREADED_HEAP	Does nothing
TRANSACTIONAL_HEAP	Needs a Transaction

The `'ExclusiveView'` function ensures an allocation is present in main memory and then claims an exclusive lock on it. The `'ExclusiveView'` function may optionally be called recursively by any thread within the *same* transaction.

The `'Address'` parameter must contain an `'Address'` returned by a previous successful call to either `'New'` or `'Resize'`.

The `'Data'` parameter will contain a pointer to the allocation in main memory if the call to `'ExclusiveView'` returns `'True'`. The `'Data'` pointer returned remains valid and the associated lock is held until a subsequent matching call is made to either `'EndExclusiveView'`, `'EndTransaction'` or `'CloseFile'`. The value of the `'Data'` pointer can vary on each call to `'ExclusiveView'` and may *not* match the value originally returned by `'New'` or `'Resize'`. The supplied pointer is a direct reference to the related section of the file stored in main memory. *There are no intermedatries or copies and it is an error to make any modifications to the data. If any changes are made to the data the outcome is undefined. If an attempt is made to read beyond the end of an allocation the outcome is also undefined.* Clearly, direct access is a very powerful double edged sword. It is certainly very fast but it also requires some degree of care.

The optional `'Space'` parameter will contain the actual number of bytes in the allocation (see `'New'` or `'Resize'`) if the call to `'ExclusiveView'` returns `'True'`.

A call to `‘ExclusiveView’` can return `‘False’` for a variety of reasons. These vary from things like a physical file transfer error to automatic deadlock detection. A call may be repeated if it fails but is likely to fail again for the same reason. In the case of deadlock, the user can simply abort the current transaction (or one of the other related transactions) by calling `‘EndTransaction’`. Simply repeating the transaction should resolve the issue in the majority of cases.

A nested call to `‘Update’` or `‘View’` after a successful call to `‘ExclusiveView’` within a transaction should always succeed as the allocation should already be in main memory and be protected by an exclusive lock. Consequently, the `‘ExclusiveView’` function can be used to minimize the calls to the more expensive `‘Update’` function. A general approach is typically to call `‘ExclusiveView’` to examine an allocation and to only call `‘Update’` if any modifications are required.

The `‘AutomaticViewScope’` class in the Rockall-DB library can be used to automatically call `‘EndExclusiveView’` at the end of a scope to make programming more straight-forward and reliable (see the `‘AutomaticViewScope’` class).

A call to `‘EndExclusiveView’`, `‘EndTransaction’` or `‘CloseFile’` will automatically release all the resources associated with a call to `‘ExclusiveView’`.

When `‘ExclusiveView’` returns `‘False’` more information is usually available by calling the `‘GetError’` function to return the error code.

5.3.6. The EndExclusiveView Function

The `‘EndExclusiveView’` function releases all the resources related to a matching call of the `‘ExclusiveView’` function within a `‘DATABASE_HEAP’` or `‘TRANSACTIONAL_HEAP’`. The `‘EndExclusiveView’` function requires an active transaction (see `‘BeginTransaction’` for more information about transactions).

A technical specification of the `‘EndExclusiveView’` function is as follows:

```
VIRTUAL BOOLEAN EndExclusiveView
  (
  FILE_ADDRESS Address,
  BOOLEAN Reset = True
  )
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	Needs a Transaction
MULTI_THREADED_HEAP	Not Supported
SINGLE_THREADED_HEAP	Not Supported
TRANSACTIONAL_HEAP	Needs a Transaction

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	Needs a Transaction
MULTI_THREADED_HEAP	Does nothing
SINGLE_THREADED_HEAP	Does nothing
TRANSACTIONAL_HEAP	Needs a Transaction

The `‘EndExclusiveView’` function will imediately release all the resources associated with a previous successful call to `‘ExclusiveView’`. Nonetheless, it may optionally allow the allocation to remain in main memory for an unspecified amout of time for performance reasons.

The `‘Address’` parameter must contain the `‘Address’` supplied by to a previous successful call to `‘ExclusiveView’`.

The `‘Reset’` parameter will reset the error code if set to `‘True’` but will leave it unchanged if set to `‘False’`. This is feature is sometimes helpful in the situation where an error occurs within nested calls and `‘EndExclusiveView’` is called during the unwinding these frames.

The `‘AutomaticViewScope’` class in the Rockall-DB library can be used to automatically call `‘EndExclusiveView’` at the end of a scope to make programming more straight-forward and reliable (see the `‘AutomaticViewScope’` class).

A call to `‘EndExclusiveView’`, `‘EndTransaction’` or `‘CloseFile’` will automatically release all the resources associated with a call to `‘ExclusiveView’`.

When `‘EndExclusiveView’` returns `‘False’` more information is usually available by calling the `‘GetError’` function to return the error code.

5.3.7. The JoinTransaction Function

The `‘JoinTransaction’` function allows multiple threads to work on a single transaction within a `‘DATABASE_HEAP’` or `‘TRANSACTIONAL_HEAP’`. A thread that calls `‘JoinTransaction’` must not currently be part of a transaction on the related heap.

A technical specification of the `‘JoinTransaction’` function is as follows:

```
VIRTUAL BOOLEAN JoinTransaction
  (
  FILE_ADDRESS TransactionID,
  TRANSACTION_LOCK<HEAP> *Lock = NULL
  )
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	No Transaction
MULTI_THREADED_HEAP	Not Supported
SINGLE_THREADED_HEAP	Not Supported
TRANSACTIONAL_HEAP	No Transaction

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	No Transaction
MULTI_THREADED_HEAP	Does nothing
SINGLE_THREADED_HEAP	Does nothing
TRANSACTIONAL_HEAP	No Transaction

The `‘JoinTransaction’` function makes the calling thread part of an active transaction. The calling thread becomes a full partner in the transaction and has access to all the resources and rights associated with the transaction.

The `‘TransactionID’` parameter must contain a `‘TransactionID’` returned by a previous successful call to `‘BeginTransaction’`.

The optional `‘Lock’` parameter will update any supplied `‘TRANSACTION_LOCK<HEAP>’` and make it refer to an internal lock associated with the transaction. The `‘Lock’` may then be used in the usual way to help manage the threads working on the transaction.

It is typically necessary to add locking code when using `‘JoinTransaction’` as any thread in the transaction will automatically get access to all the resources within the transaction. Consequently, if two threads in the same transaction both call `‘ExclusiveView’` or `‘Update’` on the same allocation both threads will be granted access (as locks in Rockall-DB are held by transactions and *not* by threads).

Any thread that is part of a transaction may call `‘EndTransaction’` but this call will not complete and will wait until all the other threads in the transaction call `‘LeaveTransaction’` before ending the transaction.

When ‘`JoinTransaction`’ returns ‘`False`’ more information is usually available by calling the ‘`GetError`’ function to return the error code.

5.3.8. The LeaveTransaction Function

The ‘`LeaveTransaction`’ function allows a thread to leave a transaction within a ‘`DATABASE_HEAP`’ or ‘`TRANSACTIONAL_HEAP`’. A thread that calls ‘`LeaveTransaction`’ must be part of a transaction on the associated heap.

A technical specification of the ‘`LeaveTransaction`’ function is as follows:

```
VIRTUAL BOOLEAN LeaveTransaction( VOID )
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	Needs a Transaction
MULTI_THREADED_HEAP	Not Supported
SINGLE_THREADED_HEAP	Not Supported
TRANSACTIONAL_HEAP	Needs a Transaction

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	Needs a Transaction
MULTI_THREADED_HEAP	Does nothing
SINGLE_THREADED_HEAP	Does nothing
TRANSACTIONAL_HEAP	Needs a Transaction

The ‘`LeaveTransaction`’ function allows a thread to leave a transaction after creating it with a call to ‘`BeginTransaction`’ or after joining it with a call to ‘`JoinTransaction`’.

All the threads associated with a transaction may leave it by calling ‘`LeaveTransaction`’. Nonetheless, at least one thread will eventually need to rejoin the transaction in order to call ‘`EndTransaction`’ to complete it.

Any thread that is part of a transaction may call ‘`EndTransaction`’. Nonetheless, this call will not complete and will wait until all the other threads in the transaction call ‘`LeaveTransaction`’ before completing it.

When ‘`LeaveTransaction`’ returns ‘`False`’ more information is usually available by calling the ‘`GetError`’ function to return the error code.

5.3.9. The Update Function

The ‘`Update`’ function allows an allocation to be transactionally updated within a ‘`DATABASE_HEAP`’ or ‘`TRANSACTIONAL_HEAP`’. The ‘`Update`’ function requires an active transaction (see ‘`BeginTransaction`’ for more information about transactions).

A technical specification of the ‘`Update`’ function is as follows:

```
VIRTUAL BOOLEAN Update
  (
  FILE_ADDRESS Address,
  VOID **Data,
  FILE_SIZE *Space = NULL
  )
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	Needs a Transaction
MULTI_THREADED_HEAP	Not Supported
SINGLE_THREADED_HEAP	Not Supported
TRANSACTIONAL_HEAP	Needs a Transaction

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	Needs a Transaction
MULTI_THREADED_HEAP	Does nothing
SINGLE_THREADED_HEAP	Does nothing
TRANSACTIONAL_HEAP	Needs a Transaction

The `'Update'` function ensures an allocation is present in main memory, claims an exclusive lock and then makes a copy of its contents. The `'Update'` function may optionally be called recursively by any thread within the *same* transaction.

The `'Address'` parameter must contain an `'Address'` returned by a previous successful call to `'New'` or `'Resize'`.

The `'Data'` parameter will contain a pointer to the allocation in main memory if the call to `'Update'` returns `'True'`. The `'Data'` pointer returned remains valid and the associated lock held until an appropriate call is made to `'EndTransaction'`. The value of the `'Data'` pointer can vary on each call to `'Update'` and may *not* match the value originally returned by `'New'` or `'Resize'`. The `'Data'` pointer may be used to modify any part of the allocation at any time up to the call of `'EndTransaction'`. The supplied pointer is a direct reference to the section of the file stored in main memory. *There are no intermedatries or copies and any changes will be reflected in the file when the buffer is written back after the transaction is complete. Obviously, writing off the end of an allocation can damage other parts of the file and/or data structures.* Clearly, direct access is a very powerful double edged sword. It is certainly very fast but it also requires some degree of care.

The optional `'Space'` parameter will contain the actual number of bytes in the allocation (see `'New'` or `'Resize'`) if the call to `'Update'` returns `'True'`.

A call to `'Update'` can return `'False'` for a variety of reasons. These vary from things like a physical file transfer error to automatic deadlock detection. A call may be repeated if it fails but is likely to fail again for the precisely same reason. In the case of deadlock, the user can abort the current transaction (or one of the related transactions) by calling `'EndTransaction'`. Simply repeating the transaction should resolve the issue in the majority of cases.

A nested call to `'ExclusiveView'` or `'View'` after a successful call to `'Update'` in the same transaction should always succeed, as the allocation will already be in main memory and be protected by an exclusive lock. The number of calls to `'Update'` should be minimize where possible as each call requires an exclusive lock and potentially a copy of the entire allocation in the database log.

A call to `'EndTransaction'` or `'CloseFile'` will automatically release all the resources associated with a call to `'Update'`.

When `'Update'` returns `'False'` more information is usually available by calling the `'GetError'` function to return the error code.

5.3.10. The View Function

The `'View'` function provides a sharable read-only view of an allocation within a `'DATABASE_HEAP'` or `'TRANSACTIONAL_HEAP'`. The `'View'` function requires an active transaction (see `'BeginTransaction'` for more information about transactions).

A technical specification of the `'View'` function is as follows:

```
VIRTUAL BOOLEAN View
  (
  FILE_ADDRESS Address,
  VOID **Data,
  FILE_SIZE *Space = NULL
  )
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	Needs a Transaction
MULTI_THREADED_HEAP	Not Supported
SINGLE_THREADED_HEAP	Not Supported
TRANSACTIONAL_HEAP	Needs a Transaction

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	Needs a Transaction
MULTI_THREADED_HEAP	Does nothing
SINGLE_THREADED_HEAP	Does nothing
TRANSACTIONAL_HEAP	Needs a Transaction

The `‘View’` function ensures an allocation is present in main memory and then claims and sharable lock on it. The `‘View’` function may optionally be called recursively by any thread within the *same* transaction.

The `‘Address’` parameter must contain an `‘Address’` returned by a previous successful call to `‘New’` or `‘Resize’`.

The `‘Data’` parameter will contain a pointer to the allocation in main memory if the call to `‘View’` returns `‘True’`. The `‘Data’` pointer returned remains valid and the associated lock is held until a subsequent matching call is made to either `‘EndView’`, `‘EndTransaction’` or `‘CloseFile’`. The value of the `‘Data’` pointer can vary on each call and may *not* match the value originally returned by `‘New’` or `‘Resize’`. The supplied pointer is a direct reference to the related section of the file stored in main memory. *There are no intermedatries or copies and it is an error to make any modifications to the data. If any changes are made to the data the outcome is undefined. If an attempt is made to read beyond the end of an allocation the outcome is also undefined.* Clearly, direct access is a very powerful double edged sword. It is certainly very fast but it also requires some degree of care.

The optional `‘Space’` parameter will contain the actual number of bytes in the allocation (see `‘New’` or `‘Resize’`) if the call to `‘View’` returns `‘True’`.

A call to `‘View’` can return `‘False’` for a variety of reasons. These vary from things like a physical file transfer error to automatic deadlock detection. A call may be repeated if it fails but is likely to fail again for the precisely same reason. In the case of deadlock, a victim must select themselves and abort their own transaction by calling `‘EndTransaction’`. Simply repeating the transaction should resolve the issue in the majority of cases.

A nested call to `‘ExclusiveView’` or `‘Update’` after a successful call to `‘View’` in the same transaction should *never* succeed, as it is not possible to upgrade a shareable lock to an exclusive lock. Consequently, a call to the `‘View’` function implies that the allocation will read but not updated. It is safe to call `‘EndView’` and subsequently call `‘ExclusiveView’` or `‘Update’` but the contents of the allocation could change between these calls.

The `‘AutomaticViewScope’` class in the Rockall-DB library can be used to automatically call `‘EndView’` at the end of a scope to make programming more straight-forward and reliable (see the `‘AutomaticViewScope’` class).

A call to `‘EndView’`, `‘EndTransaction’` or `‘CloseFile’` will automatically release all the resources associated with a call to `‘View’`.

When `‘View’` returns `‘False’` more information is usually available by calling the `‘GetError’` function to return the error code.

5.3.11. The EndView Function

The `‘EndView’` function releases all the resources related to a matching call of the `‘View’` function within a `‘DATABASE_HEAP’` or `‘TRANSACTIONAL_HEAP’`. The `‘EndView’` function requires an active transaction (see `‘BeginTransaction’` for more information about transactions).

A technical specification of the `‘EndView’` function is as follows:

```
VIRTUAL BOOLEAN EndView
  (
  FILE_ADDRESS Address,
  BOOLEAN Reset = True
  )
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	Needs a Transaction
MULTI_THREADED_HEAP	Not Supported
SINGLE_THREADED_HEAP	Not Supported
TRANSACTIONAL_HEAP	Needs a Transaction

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	Needs a Transaction
MULTI_THREADED_HEAP	Does nothing
SINGLE_THREADED_HEAP	Does nothing
TRANSACTIONAL_HEAP	Needs a Transaction

The `‘EndView’` function will imediately release all the resources associated with a previous successful call to `‘View’`. Nonetheless, it may optionally allow the allocation to remain in main memory for an unspecified amout of time for performance reasons.

The `‘Address’` parameter must contain an `‘Address’` returned by a previous successful call to `‘View’`.

The `‘Reset’` parameter will reset the error code if set to `‘True’` but will leave it unchanged if set to `‘False’`. This is feature is sometimes helpful in the situation where an error occurs within nested calls and `‘EndView’` is called during the unwinding these frames.

The `‘AutomaticViewScope’` class in the Rockall-DB library can be used to automatically call `‘EndView’` at the end of a scope to make programming more straight-forward and reliable (see the `‘AutomaticViewScope’` class).

A call to `‘EndView’`, `‘EndTransaction’` or `‘CloseFile’` will automatically release all the resources associated with a call to `‘View’`.

When `‘EndView’` returns `‘False’` more information is usually available by calling the `‘GetError’` function to return the error code.

5.3.12. The EndTransaction Function

The `‘EndTransaction’` function ends a transaction created by a previous call to `‘BeginTransaction’` for a `‘DATABASE_HEAP’` or `‘TRANSACTIONAL_HEAP’`. There may *only be one active transaction per thread* at any time (see `‘AutomaticHeapScope’` for support of nested transactions).

A technical specification of the `‘EndTransaction’` function is as follows:

```
VIRTUAL BOOLEAN EndTransaction
  (
  BOOLEAN Abort = False,
  BOOLEAN Commit = False
  )
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	Needs a Transaction
MULTI_THREADED_HEAP	Not Supported
SINGLE_THREADED_HEAP	Not Supported
TRANSACTIONAL_HEAP	Needs a Transaction

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	Needs a Transaction
MULTI_THREADED_HEAP	Does nothing
SINGLE_THREADED_HEAP	Does nothing
TRANSACTIONAL_HEAP	Needs a Transaction

The `'EndTransaction'` function releases all the resources associated with a transaction and terminates it. Any calls to `'ExclusiveView'`, `'New'`, `'Resize'`, `'Update'` or `'View'` on a `'DATABASE_HEAP'` or `'TRANSACTIONAL_HEAP'` are also terminated as part of this process. Any `'Data'` pointers returned by any of these calls are no longer valid and any associated locks are released.

The `'Abort'` parameter drastically changes the effects of the `'EndTransaction'` function. If it is set to `'False'` any updates are accepted and any call to `'Delete'` is executed and the associated allocations removed from a `'DATABASE_HEAP'` or `'TRANSACTIONAL_HEAP'`. If it is set to `'True'` any calls to `'Delete'`, `'New'`, `'Resize'`, `'SetUserValue'` or `'Update'` are undone on a `'DATABASE_HEAP'` or `'TRANSACTIONAL_HEAP'`.

The `'Commit'` parameter only applies to a `'DATABASE_HEAP'`. If it is set to `'False'` the transaction will be committed asynchronously on the next commit cycle. The commit cycle occurs regularly but it is possible that the whole transaction might be lost if the system fails before this occurs. All transactions are committed in strict order of completion and so this should not be an issue, as such a failure should only lose the last few transactions. There is a huge performance advantage of setting `'Commit'` to `'False'` as the calling thread is immediately available for other work, as it does not need to wait for any physical file transfer to complete. If `'Commit'` is set to `'True'` the transaction will be committed synchronously and the thread will wait until all completed transactions up to the point of the call on any thread have been committed to the file. In rare situations (i.e. say 1 GB updates), this may take a number of seconds to complete reducing the number of processing threads available for work and so seriously impact the overall performance.

The `'AutomaticHeapScope'` class in the Rockall-DB library can be used to automatically call `'EndTransaction'` at the end of a scope to make programming more straight-forward and reliable (see the `'AutomaticHeapScope'` class).

A call to `'CloseFile'` will also release all the resources associated with a call to `'BeginTransaction'` as it aborts all active transactions.

When `'EndTransaction'` returns `'False'` more information is usually available by calling the `'GetError'` function to return the error code.

5.3.13. The DeleteRegion Function

The `'DeleteRegion'` function is intended for advanced users and deletes a sub-heap within a `'DATABASE_HEAP'` or `'TRANSACTIONAL_HEAP'`. When `'DeleteRegion'` is called there must not be any active transactions on the heap.

A technical specification for `'DeleteRegion'` is as follows:

```
VIRTUAL BOOLEAN DeleteRegion
  (
  SBIT32 RegionID
  )
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	No Transaction
MULTI_THREADED_HEAP	Not Supported
SINGLE_THREADED_HEAP	Not Supported
TRANSACTIONAL_HEAP	No Transaction

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	No Transaction
MULTI_THREADED_HEAP	Does nothing
SINGLE_THREADED_HEAP	Does nothing
TRANSACTIONAL_HEAP	No Transaction

The `'DeleteRegion'` function is part of a family of related functions. These are `'CreateRegion'`, `'ChangeRegion'` and `'DeleteRegion'`. There is no limit to the number of regions that can be created within a `'DATABASE_HEAP'` or `'TRANSACTIONAL_HEAP'`. A region is very much like a separate sub-heap within a heap.

The `'RegionID'` parameter must contain the `'ID'` of an existing region returned by a previous successful call to `'CreateRegion'`.

When a call to `'DeleteRegion'` is made all of the contents of the region are deleted in much the same way as a call to `'DeleteAll'` and then the region itself is deleted.

A call to `'DeleteRegion'` is not transactional but is atomic and so all or none of the region will be deleted.

When `'DeleteRegion'` returns `'False'` more information is usually available by calling the `'GetError'` function to return the error code.

5.3.14. The CloseFile Function

The `'CloseFile'` function closes a `'DATABASE_HEAP'` or `'TRANSACTIONAL_HEAP'`. When `'CloseFile'` is called there should not be any active transactions on the heap.

A technical specification of the `'CloseFile'` function is as follows:

```
VIRTUAL BOOLEAN CloseFile( VOID )
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	No Transaction
MULTI_THREADED_HEAP	Not Supported
SINGLE_THREADED_HEAP	Not Supported
TRANSACTIONAL_HEAP	No Transaction

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	No Transaction
MULTI_THREADED_HEAP	Does nothing
SINGLE_THREADED_HEAP	Does nothing
TRANSACTIONAL_HEAP	No Transaction

The `'CloseFile'` function closes a `'DATABASE_HEAP'` or `'TRANSACTIONAL_HEAP'`. There should be no active transactions on the heap and it should be idle. If there are active transactions an attempt will be made to automatically abort them so that all future calls to the heap will fail. Regardless, it is fundamentally dangerous if the heap is not idle as any active threads may have `'Data'` pointers to areas in the heap and so may crash as resources are deleted during clean up.

A call to `'CloseFile'` on a `'DATABASE_HEAP'` may take some time to complete as it implies waiting for any active threads to complete and synchronously flushing all changes back to the file.

A call to `'CloseFile'` on a `'TRANSACTIONAL_HEAP'` deletes everything on the heap in a similar way to calling `'DeleteRegion'` on all the regions.

When `'CloseFile'` returns `'False'` more information is usually available by calling the `'GetError'` function to return the error code.

5.4. The Rockall-DB Database Functions

The Rockall-DB database functions are only available in a Rockall-DB '`DATABASE_HEAP`' (see section 5.1 for a review of the Rockall-DB heaps). A table of the Rockall-DB functions covered in this section is as follows:

Function Name	Description
`CreateShadowFile`	Create a bitwise replica of an active `'DATABASE_HEAP'` file and keep it up-to-date.
`CreateShapshotFile`	Create a stand-alone bitwise replica of an active `'DATABASE_HEAP'` file.
`OpenFile`	Open an existing `'DATABASE_HEAP'` file and prepare it for active use.
`Commit`	Wait until all completed transactions have been written to file.
`FileSize`	Get the committed file size.
`GetUserValue`	Get a user value previously stored by `'SetUserValue'`.
`SetUserValue`	Store a user value in a reserved area within a `'DATABASE_HEAP'` file.
`Touch`	Asynchronously bring to memory an allocation previously created by `'New'` or `'Resize'`.
`CloseShadowFile`	Close a shadow file (i.e. the converse of `'CreateShadowFile'`).

Table-05-05

All of the functions listed in 'Table-05-05' are described in detail in the following sections in the order they appear the table above.

5.4.1. The CreateShadowFile Function

The `'CreateShadowFile'` function creates a bitwise replica of a `'DATABASE_HEAP'` file and keeps it up-to-date. The `'CreateShadowFile'` function can be called regardless of whether there is an active transaction on the Rockall-DB heap.

A technical specification of the variants of the `'CreateShadowFile'` function is as follows:

```
VIRTUAL BOOLEAN CreateShadowFile
  (
  CHAR *FileName,
  SBIT32 *ID = NULL,
  BOOLEAN Wait = True
  )

VIRTUAL BOOLEAN CreateShadowFile
  (
  WCHAR *FileName,
  SBIT32 *ID = NULL,
  BOOLEAN Wait = True
  )
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	Any State
MULTI_THREADED_HEAP	Not Supported
SINGLE_THREADED_HEAP	Not Supported
TRANSACTIONAL_HEAP	Not Supported

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	Any State
MULTI_THREADED_HEAP	Does nothing
SINGLE_THREADED_HEAP	Does nothing
TRANSACTIONAL_HEAP	Does nothing

The `'CreateShadowFile'` function comes in both 'ACSII' and 'Unicode' styles. A call to `'CreateShadowFile'` can only be made after a successful call to either `'CreateFile'` or `'OpenFile'`. All of the parameters are optional except for the `'FileName'` parameter which is mandatory.

The `'FileName'` parameter is the name of a file. If the file does not exist a new file will be created and the contents of the file opened by the call to `'CreateFile'` or

`'OpenFile'` will be copied into it. If the file already exists and the database header information is *identical* to the file opened by the call to `'CreateFile'` or `'OpenFile'` then no copying will be done. However, if the file already exists and the header information is *not identical* then the contents of the file will be truncated and the entire file rewritten and bought up-to-date. Regardless, once a shadow file is active and up-to-date all new transactions will be replicated into it. Consequently, a shadow file should always be a bitwise copy of the master file, except during commit where it may lag by a few milliseconds. Any shadow file successfully closed by a call to ‘`CloseShadowFile`’ or ‘`CloseFile`’ should (at the instant it is closed) be an identical bitwise copy of the master file.

The `'ID'` parameter, if specified, will contain a unique `'ID'` which can be used in a later call to `'CloseShadowFile'`. At the time of writing the `'ID'` is always in the range of 0 to 3, as there is currently a limit of 4 concurrently active shadow files.

The `'Wait'` parameter, if specified, controls whether the caller will wait for any copying of the shadow file to complete or whether it will return immediately.

A shadow file is a full member of a Rockall-DB `'DATABASE_HEAP'` and is kept up-to-date by the logger as part of the standard commit processing. Consequently, if a shadow file has an error Rockall-DB will delay the commit and repeatedly try to rewrite the data until it is successful. If a shadow file is slow for any reason then Rockall-DB will wait for it to catch up. Consequently, the performance and reliability of a shadow file can affect the overall performance of a Rockall-DB database. Therefore, it is seldom wise to put any Rockall-DB shadow file onto busy or low performing storage (i.e. like a USB stick or low performance remote storage) as this may significantly increase the time it takes to commit transactions.

When `'CreateShadowFile'` returns `'False'` more information is usually available by calling the `'GetError'` function to return the error code.

5.4.2. The CreateSnapshotFile Function

The `'CreateSnapshotFile'` function creates a bitwise replica of a `'DATABASE_HEAP'` file. The `'CreateSnapshotFile'` function can be called regardless of whether there is an active transaction on the Rockall-DB heap.

A technical specification of the variants of the `'CreateShadowFile'` function is as follows:

```
VIRTUAL BOOLEAN CreateSnapshotFile
  (
  CHAR *FileName
  )

VIRTUAL BOOLEAN CreateSnapshotFile
  (
  WCHAR *FileName
  )
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	Any State
MULTI_THREADED_HEAP	Not Supported
SINGLE_THREADED_HEAP	Not Supported
TRANSACTIONAL_HEAP	Not Supported

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	Any State
MULTI_THREADED_HEAP	Does nothing
SINGLE_THREADED_HEAP	Does nothing
TRANSACTIONAL_HEAP	Does nothing

The `'CreateSnapshotFile'` function comes in both ‘ACSII’ and ‘Unicode’ styles. A call to `'CreateSnapshotFile'` can only be made after a successful call to either

‘`CreateFile`’ or ‘`OpenFile`’. A call to ‘`CreateSnapshotFile`’ is functionally equivalent to calling ‘`CreateShadowFile`’ followed by ‘`CloseShadowFile`’.

The file created by ‘`CreateSnapshotFile`’ should be an exact replica of the file that was opened by the call to ‘`CreateFile`’ or ‘`OpenFile`’ at the instant the call to ‘`CreateSnapshotFile`’ returns. All the same considerations apply to ‘`CloseSnapshotFile`’ as apply to ‘`CloseShadowFile`’ and there is a possibility that the new file may contain partial transactions that will be recovered when the snapshot is opened later by a call to ‘`OpenFile`’ (see ‘`CloseShadowFile`’ for details).

When ‘`CreateSnapshotFile`’ returns ‘`False`’ more information is usually available by calling the ‘`GetError`’ function to return the error code.

5.4.3. The OpenFile Function

The ‘`OpenFile`’ function prepares an existing ‘`DATABASE_HEAP`’ file for use. When ‘`OpenFile`’ is called there must not be any active transactions on the heap.

A technical specification of the variants of the ‘`OpenFile`’ function are as follows:

Rockall-DB Heap	***Comments***
DATABASE_HEAP	No Transaction
MULTI_THREADED_HEAP	Not Supported
SINGLE_THREADED_HEAP	Not Supported
TRANSACTIONAL_HEAP	Not Supported

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	No Transaction
MULTI_THREADED_HEAP	Does nothing
SINGLE_THREADED_HEAP	Does nothing
TRANSACTIONAL_HEAP	Does nothing

```
VIRTUAL BOOLEAN OpenFile
  (
  CHAR *FileName,
  BOOLEAN ReadOnly = False,
  ENCRYPTION_FUNCTION Function = NULL,
  VOID *UserValue = NULL
  )

VIRTUAL BOOLEAN OpenFile
  (
  WCHAR *FileName,
  BOOLEAN ReadOnly = False,
  ENCRYPTION_FUNCTION Function = NULL,
  VOID *UserValue = NULL
  )

typedef VOID (*ENCRYPTION_FUNCTION)
  (
  FILE_ADDRESS    Address,
  VOID            *Data,
  BOOLEAN         Decrypt,
  BOOLEAN         Encrypt,
  FILE_SIZE       Size,
  VOID            *UserValue
  )
```

The ‘`OpenFile`’ function comes in both ‘ACSII’ and ‘Unicode’ styles. All of the parameters are optional except for the ‘`FileName`’ parameter which is mandatory.

The ‘`FileName`’ parameter must contain the path to an existing Rockall-DB database file. The file must *already exist* and have been previously created by a call to ‘`CreateFile`’. The contents and the format of the file are checked to ensure the file meets the necessary requirements. If the file contains any partial transactions these will be undone if ‘`ReadOnly`’ is set to ‘`False`’. The transactions will be undone as if they had been terminated with a call to ‘`EndTransaction`’ with the ‘`Abort`’ parameter set to `'True'`. If any of these requirements are not met the ‘`OpenFile`’ function will return `'False'`.

The optional `'ReadOnly'` parameter controls how the file is opened. If the `'ReadOnly'` parameter is set to `'False'` then the file will be opened for reading and writing and the contents of the file will be recovered if needed. The file will be locked so that any other attempts to open the file will fail. If the `'ReadOnly'` parameter is set to `'True'` then the file will be opened for just reading and the call will fail if any recovery is needed. The file will be locked so that further attempts to open the file for reading will succeed but attempts to open the file for writing will fail.

The optional `'Function'` parameter allows a user supplied ‘`ENCRYPTION_FUNCTION`’ to be supplied which will be called by Rockall-DB every time a data block is read or written to the file. When a block is read the ‘`ENCRYPTION_FUNCTION`’ will be called just after the read completes and it is required to immediately decrypt it. When a block is written the ‘`ENCRYPTION_FUNCTION`’ will be called just before the write starts and the it is required to immediately encrypt it. The meaning of the parameters supplied to the ‘`ENCRYPTION_FUNCTION`’ is as follows:

The `'Address'` parameter is the offset in the file (in bytes) of the start of block.

The `'Data'` parameter is the memory address of the data to be encrypted or decrypted.

The `'Decrypt'` parameter will be `'True'` if the `'Data'` needs to be decrypted just after it has been read otherwise it will be `'False'`.

The `'Encrypt'` parameter will be `'True'` if the `'Data'` needs to be encrypted just before a write otherwise it will be `'False'`.

The `'Size'` parameter contains the number of bytes in the area to be encrypted or decrypted.

The `'UserValue'` parameter contains the `'UserValue'` parameter initially passed to `'OpenFile'`. This enables a user supplied value (or a pointer to a defined user structure) to be passed to all invokations of the ‘`ENCRYPTION_FUNCTION`’.

Technically, the ‘`ENCRYPTION_FUNCTION`’ does not actually need to decrypt and encrypt the data. Instead, it could use the opportunity to do some other form of processing on the data. Regardless, damaging the data in any way may lead to grave disorder in Rockall-DB and will likely corrupt the database.

When `'OpenFile'` returns `'False'` more information is usually available by calling the `'GetError'` function to return the error code.

5.4.4. The Commit Function

The `'Commit'` function will wait until ***all*** completed transactions have been written to the `'DATABASE_HEAP'` file. The `'Commit'` function can be called regardless of whether there is an active transaction on the Rockall-DB heap.

A technical specification of the `'Commit'` function is as follows:

```
VIRTUAL BOOLEAN Commit( VOID )
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	Any State
MULTI_THREADED_HEAP	Not Supported
SINGLE_THREADED_HEAP	Not Supported
TRANSACTIONAL_HEAP	Not Supported

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	Any State
MULTI_THREADED_HEAP	Does nothing
SINGLE_THREADED_HEAP	Does nothing
TRANSACTIONAL_HEAP	Does nothing

The `'Commit'` function will not return until *all* completed transactions have been stored in the `'DATABASE_HEAP'` file and in all related shadow files.

If `'Commit'` is *not* called then the last few completed transactions can sometimes be lost if the system fails before the automatic `'Commit'` completes. All transactions are written in strict order of completion so later transactions can never be kept if earlier transactions are lost. Typically, this is all that is needed in most situations and so there is seldom a need to call `'Commit'`. Nonetheless, if a thread needs to be certain that a specific transaction has commited then `'Commit'` will not return until this is the case.

If a call to `'Commit'` is made then logging will commence immediately, if not already in progress. A call to `'Commit'` does not affect any other functionality in Rockall-DB as the call merely waits while all the outstanding transactions are written to the `'DATABASE_HEAP'` file. Occasionally, this may take a few seconds if the number or size of the changes is large (i.e. an `'Update'` to an allocation of 1GB). Moreover, it is prudent to minimise the number of calls to `'Commit'` as far as possible and only call it when absolutely necessary.

When `'Commit'` returns `'False'` more information is usually available by calling the `'GetError'` function to return the error code.

5.4.5. The FileSize Function

The `'FileSize'` function returns the committed size of the `'DATABASE_HEAP'` file. The `'FileSize'` function can be called regardless of whether there is an active transaction on a Rockall-DB heap.

A technical specification of the `'FileSize'` function is as follows:

```
VIRTUAL BOOLEAN FileSize
   (
   FILE_SIZE *Size
   )
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	Any State
MULTI_THREADED_HEAP	Not Supported
SINGLE_THREADED_HEAP	Not Supported
TRANSACTIONAL_HEAP	Not Supported

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	Any State
MULTI_THREADED_HEAP	Does nothing
SINGLE_THREADED_HEAP	Does nothing
TRANSACTIONAL_HEAP	Does nothing

The `'Size'` parameter will contain the 'committed file size' in bytes if a call to `'FileSize'` returns `'True'`. The 'committed file size' is not the actual size of the file but rather the size it would be if it was immediately closed by a call to `'CloseFile'`.

A Rockall-DB file typically contains a certain degree of padding while it is in use. This padding is automatically truncated when the file is closed by a call to `‘CloseFile’`. Consequently, there is often a significant delta between the ‘actual file size’ and the ‘committed file size’ during processing.

When `‘FileSize’` returns `‘False’` more information is usually available by calling the `‘GetError’` function to return the error code.

5.4.6. The GetUserValue Function

The `‘GetUserValue’` function will return a value previously stored by a call to `‘SetUserValue’` in the reserved area of a `‘DATABASE_HEAP’` file. The `‘GetUserValue’` function can be called regardless of whether there is an active transaction on the Rockall-DB heap.

A technical specification of the `‘GetUserValue’` function is as follows:

```
VIRTUAL BOOLEAN GetUserValue
  (
  SBIT32 ID,
  FILE_ADDRESS *Value
  )
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	Any State
MULTI_THREADED_HEAP	Not Supported
SINGLE_THREADED_HEAP	Not Supported
TRANSACTIONAL_HEAP	Not Supported

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	Any State
MULTI_THREADED_HEAP	Does nothing
SINGLE_THREADED_HEAP	Does nothing
TRANSACTIONAL_HEAP	Does nothing

The `‘GetUserValue’` function is part of a mechanism which allows users to reliably locate specific allocations within a `‘DATABASE_HEAP’` file. This mechanism has been designed so that it is still available even if Rockall-DB is not in use.

The `‘ID’` parameter is a unique user supplied value which identifies the value being loaded. It is recommended that the value of an `‘ID’` should be small (i.e. less than 16) but there are no actual restrictions besides that it must be positive.

The `‘Value’` parameter will contain any value previously stored by a call to `‘SetUserValue’` if a call to `‘GetUserValue’` returns `‘True’`. If there was no previous call to `‘SetUserValue’` it will contain zero. Typically, a `‘Value’` will be a `‘FILE_ADDRESS’` (i.e. `‘Address’`) returned by a previous call to `‘New’` or `‘Resize’`. The `‘Value’` can then be used in subsequent calls to `‘ExclusiveView’`, `‘Update’` or `‘View’` to load the associated allocation into memory. Typically, this would be the root of a more complex data structure containing other `‘FILE_ADDRESS’` (i.e. `‘Address’`) values which could then be accessed using the same steps as outlined above.

The `‘GetUserValue’` and `‘SetUserValue’` functions are typically most useful when opening an existing `‘DATABASE_HEAP’` file. Here, it is often necessary to find the root of an existing data structure in the file. The `‘GetUserValue’` and `‘SetUserValue’` functions are intended to make this easy by allowing one or more arbitrary `‘FILE_ADDRESS’` (i.e. `‘Address’`) values to be stored in the specially reserved area within the `‘DATABASE_HEAP’` file.

Technically, it is not necessary to call `‘GetUserValue’` to read a stored user value. The location of *all* the user values is always at the `‘Address’` pointed to by the first eight bytes (i.e. a `‘FILE_ADDRESS’`) in a `‘DATABASE_HEAP’` file. Let’s call this value the `‘BASE’` address. A specific user value with an `‘ID’` of `‘X’` is always stored at `‘BASE+(X*sizeof(FILE_ADDRESS))’` within the `‘DATABASE_HEAP’` file. Consequently, any user value in a `‘DATABASE_HEAP’` can be found without any use of Rockall-DB. Furthermore, as the `‘Address’` returned by calls to `‘New’` or `‘Resize’` are also of type `‘FILE_ADDRESS’` it is also possible to navigate between allocations without using Rockall-DB in the same way.

Just for completeness, it should be noted that the second eight bytes in a `‘DATABASE_HEAP’` file contain the `‘ByteSex’` of the `‘DATABASE_HEAP’` file. If this value is loaded it should be equal to the constant ‘0x0001020304050607’. If not, then the `‘ByteSex’` of the current machine is different from the machine that originally created the `‘DATABASE_HEAP’` file. Consequently, all the `‘FILE_ADDRESS’` values (and any other data with the incorrect `‘ByteSex’`) will need to be converted to the correct `‘ByteSex’` before use.

The ability to navigate a `‘DATABASE_HEAP’` file without using Rockall-DB makes it truly ‘open’, to a degree seldom seen in database products. Certainly, there is no need to be concerned about propriety formats or being ‘locked-in’ with Rockall-DB, as user data can always be directly accessed (i.e. as outlined above). Consequently, Rockall-DB is a good tool for designing and building open standards, as it is easy to use and allows its presence to be hidden behind an alternative file navigation mechanism.

When `‘GetUserValue’` returns `‘False’` more information is usually available by calling the `‘GetError’` function to return the error code.

5.4.7. The SetUserValue Function

The `‘SetUserValue’` function stores a user supplied value in a reserved area of a `‘DATABASE_HEAP’` file. The `‘SetUserValue’` function requires an active transaction (see `‘BeginTransaction’` for more information about transactions).

A technical specification of the `‘SetUserValue’` function is as follows:

```
VIRTUAL BOOLEAN SetUserValue
  (
  SBIT32 ID,
  FILE_ADDRESS Value
  )
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	Needs a Transaction
MULTI_THREADED_HEAP	Not Supported
SINGLE_THREADED_HEAP	Not Supported
TRANSACTIONAL_HEAP	Not Supported

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	Needs a Transaction
MULTI_THREADED_HEAP	Does nothing
SINGLE_THREADED_HEAP	Does nothing
TRANSACTIONAL_HEAP	Does nothing

The `‘SetUserValue’` function is part of a mechanism which allows users to reliably locate specific allocations within a `‘DATABASE_HEAP’` file. This mechanism has been designed so that it is still available even if Rockall-DB is not in use.

The `‘ID’` parameter is a unique user supplied value which identifies the value being stored. It is recommended that the value of an `‘ID’` should be small (i.e. less than 16) but there are no actual restrictions besides that it must be positive. When a call is made to `‘GetUserValue’` with the same `‘ID’` the value stored by the call to `‘SetUserValue’` will be returned.

The type of the `‘Value’` parameter is a `‘FILE_ADDRESS’` but it may contain any compatible value. Typically, a `‘Value’` would be a `‘FILE_ADDRESS’` (i.e. `‘Address’`) returned by a previous call to `‘New’` or `‘Resize’` referring to an allocation at the root of a more complex data structure (see `‘GetUserValue’` above).

The `‘GetUserValue’` and `‘SetUserValue’` functions are typically most useful when opening an existing `‘DATABASE_HEAP’` file. Here, it is often necessary to find the root of an existing data structure in the file. The `‘GetUserValue’` and `‘SetUserValue’` functions are intended to make this easy by allowing one or more arbitrary `‘FILE_ADDRESS’` (i.e. `‘Address’`) values to be stored in the specially reserved area within the `‘DATABASE_HEAP’` file.

When `‘SetUserValue’` returns `‘False’` more information is usually available by calling the `‘GetError’` function to return the error code.

5.4.8. The Touch Function

The `‘Touch’` function ensures an allocation returned by a previous call to `‘New’` or `‘Resize’` is already in main memory. If not, it will try to asynchronously load it into main memory from the associated `‘DATABASE_HEAP’` file. The `‘Touch’` function can be called regardless of whether there is an active transaction on a Rockall-DB heap.

A technical specification of the `‘Touch’` function is as follows:

```
VIRTUAL BOOLEAN Touch
   (
   FILE_ADDRESS Address
   )
```

Rockall-DB Heap	***Comments***
DATABASE_HEAP	Any State
MULTI_THREADED_HEAP	Not Supported
SINGLE_THREADED_HEAP	Not Supported
TRANSACTIONAL_HEAP	Not Supported

AUTOMATIC_HEAP_SCOPE	***Comments***
DATABASE_HEAP	Any State
MULTI_THREADED_HEAP	Does nothing
SINGLE_THREADED_HEAP	Does nothing
TRANSACTIONAL_HEAP	Does nothing

The `‘Address’` parameter should contain a `‘FILE_ADDRESS’` returned by a previous call to `‘New’` or `‘Resize’`. The associated page (i.e. typically 64k) will be asynchronously loaded into main memory if it is not already present.

The `‘Touch’` function can significantly improve the overall performance of calls such as `‘Delete’`, `‘ExclusiveView’`, `‘Resize’`, `‘Update’` and `‘View’` in a `‘DATABASE_HEAP’` when the associated allocation is not already in main memory. A call to `‘Touch’` must be made *well-ahead* of any expected use, so as to give the I/O sub-system plenty of time to read the associated page from the file. If not, then the overheads of a call to `‘Touch’` will usually exceed any of the potential benefits.

It is not necessary to call `‘Touch’` more than once per page (i.e. typically 64k) in Rockall-DB. Consequently, if multiple `‘Address’` values fall within a single page then only one call to `‘Touch’` is necessary to load all the related allocations.

A large number of calls to `‘Touch’` can sometimes exceed the I/O subsystems ability to service the requests. There are no restrictions in Rockall-DB and it is easily possible to make the I/O subsystem scream with pain. Under such circumstances, it is unlikely that other processes will receive good I/O service and this may lead to other issues. Under extreme load, `‘Touch’` will move to synchronous mode when it runs out of resources. Consequently, some thought is needed when using `‘Touch’`, especially in regards to large data sets (i.e. say over 1,000 pages or 200MB).

A call to `‘Touch’` has similar cost as a call to `‘ExclusiveView’` or `‘View’` and so should not be made carelessly. It should be viewed as an investment of CPU time to reduce I/O wait time. A good example of using `‘Touch’` is provided in the `‘ROW_SET’` class in the Rockall-DB library.

When `‘Touch’` returns `‘False’` more information is usually available by calling the `‘GetError’` function to return the error code.

5.4.9. The CloseShadowFile Function

The `‘CloseShadowFile’` function closes a shadow file previously created by a call to `‘CreateShadowFile’`. The `‘CloseShadowFile’` function can be called regardless of whether there is an active transaction on the Rockall-DB heap.

A technical specification of the `‘CloseShadowFile’` function is as follows:

```
VIRTUAL BOOLEAN CloseShadowFile
   (
   SBIT32 ID = 0
   )
```

Rockall-DB Heap	*Comments*
DATABASE_HEAP	Any State
MULTI_THREADED_HEAP	Not Supported
SINGLE_THREADED_HEAP	Not Supported
TRANSACTIONAL_HEAP	Not Supported

AUTOMATIC_HEAP_SCOPE	*Comments*
DATABASE_HEAP	Any State
MULTI_THREADED_HEAP	Does nothing
SINGLE_THREADED_HEAP	Does nothing
TRANSACTIONAL_HEAP	Does nothing

The `‘ID’` parameter should be the `‘ID’` returned from a previous call to `‘CreateShadowFile’`. The default value is zero as this is the first `‘ID’` returned by `‘CreateShadowFile’` and so is valid if there is only one active shadow file at a time.

A shadow file is closed after a commit is complete. Nonetheless, there may still be incomplete transactions within a shadow file. Any such transactions will be automatically rolled back and the file recovered the first time it is opened by a call to `‘OpenFile’` with the `‘ReadOnly’` parameter is set to `‘False’`.

A call to `‘OpenFile’` on a shadow file with the `‘ReadOnly’` parameter is set to `‘True’` can fail as it may not be possible to undo any partial transactions. In this case, a successful call to `‘OpenFile’` with the `‘ReadOnly’` parameter is set to `‘False’` should be made followed by a call to `‘CloseFile’`. A call to `‘OpenFile’` with the `‘ReadOnly’` parameter is set to `‘True’` should then succeed as any recovery should have taken place.

When `‘CloseShadowFile’` returns `‘False’` more information is usually available by calling the `‘GetError’` function to return the error code.

6. The Rockall-DB Library

The Rockall-DB library is a collection of classes built on top of the Rockall-DB core (as described in chapter 3 and chapter 5 above). The Rockall-DB library contains a range of functionality, such as hash tables, queues, sets, stacks, strings and trees (i.e. indexes), with or without locks, which work on all of the Rockall-DB heaps. The library is provided in source code as a collection of C++ template classes. Consequently, the Rockall-DB library is not only useful as a tool-set but also as a collection of practical examples.

In the following sections we will examine all of the classes in the Rockall-DB library and briefly describe their use.

6.1. The Rockall-DB Library Core

A number of the source files in the Rockall-DB library are part to the Rockall-DB core (as described in chapter 3 and chapter 5 above) and are as follows:

File Name	Chapters	Description
DatabaseHeap.hpp	3 & 5.1	A multi-threaded in file transactional heap
MultiThreadedHeap.hpp	3 & 5.1	A multi-threaded in memory heap.
SingleThreadedHeap.hpp	3 & 5.1	A single threaded in memory heap.
TransactionalHeap.hpp	3 & 5.1	A multi-threaded in memory transactional heap.
VirtualHeap.hpp	3	The virtual base class for all Rockall-DB heaps.

Table-06-01

Additionally, a number of other files are only intended for internal use and so are not documented in this manual or directly supported, except in regards to the wider product. A list of these file is provided in the table below:

File Name	Chapters	Description
BaseTypes.hpp	N/A	A collection of internal type names.
CompilerFlags.hpp	N/A	A collection of internal compiler related flags.
Error.hpp	N/A	A collection of Rockall-DB error codes.
ExportFailure.hpp	N/A	An internal class for exporting `'FAILURE'`.
ExportFileLock.hpp	N/A	An internal class for exporting `'FILE_LOCK'`.
ExportHeap.hpp	N/A	An internal class for exporting the heap interface.
ExportMemoryLock.hpp	N/A	An internal class for exporting `'MEMORY_LOCK'`.
ExportThreadLocalStore.hpp	N/A	An internal class for exporting `'TLS'` memory.
Failure.hpp	N/A	An internal class to support `'FAILURE'`.
MemoryFunctions.hpp	N/A	A collection internal memory copying functions.
PublicConstants.hpp	N/A	A collection of internal public constants.
ReservedWords.hpp	N/A	A collection of internal reserved words.
String.hpp	N/A	A base class related to strings.
StringFunctions.hpp	N/A	A collection internal string functions.
SubSet.hpp	N/A	A support class related to sets.
SubString.hpp	N/A	A support class related to strings.
VirtualThreadLocalStore.hpp	N/A	An internal class to support `'TLS'` memory.

Table-06-02

6.2. The Rockall-DB Library Data Structures

A number of the files in the Rockall-DB library implement classical software data structures and are as follows:

File Name	Chapters	Description
AutomaticHeapScope.hpp	6.2.1.	The heart of the Rockall-DB library.
FixedString.hpp	6.2.2.	A class for fixed length strings.
FlexibleString.hpp	6.2.3.	A class for variable length strings.
Hash.hpp	6.2.4.	A class for hash tables (i.e. lookup tables).
Queue.hpp	6.2.5.	A class for queues (i.e. FIFO tables).
RowSet.hpp	6.2.6.	A class for grouping row sets (i.e. RIDs).
Set.hpp	6.2.7.	A class for storing a sorted set of values.
Stack.hpp	6.2.8.	A class for stacks (i.e. LIFO tables).
Tree.hpp	6.2.9.	A class for multi-way trees (i.e. indexes).

Table-06-03

All of the classes listed in ‘Table-06-03’ use the `‘AutomaticHeapScope.hpp’` class as a foundation. Consequently, it is suggested that the information relating to this class should be fully understood before moving on to any of the other classes listed in this table.

6.2.1. The AutomaticHeapScope Class

The `‘AUTOMATIC_HEAP_SCOPE’` class provides a unified methodology for using all of the Rockall-DB heaps and is used throughout the Rockall-DB library. An `‘AUTOMATIC_HEAP_SCOPE’` can be easily created as shown in ‘Example-06-01’ below:

```
#include "AutomaticHeapScope.hpp"
#include "SingleThreadedHeap.hpp"
#include "Stack.hpp"

typedef SINGLE_THREADED_HEAP HEAP_TYPE;

int main( void )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope1( & Heap );
  STACK<HEAP_TYPE,NO_LOCK,int> Stack;
  int Result = 1;

  if ( Stack.PushValue( 0 ) )
    { Result = 0; }

  return Result;
  }
```

Example-06-01

The `‘AUTOMATIC_HEAP_SCOPE’` class exists because it is not possible to pass parameters to every kind of function in C++ (or related languages like C# and Java). In particular, it is not possible to pass parameters to destructors in most implementations of C++. Obviously, this is a serious issue if a destructor needs to delete an object but does not know which heap to use. It is not possible to use pointers as an object may be stored in a file (i.e. a `‘DATABASE_HEAP’`). Consequently, a consistent mechanism is needed to ensure that functions can always find the appropriate heap and any related data.

The `‘STACK’` class in ‘Example-06-01’ uses the `‘Heap’` supplied to the `‘AUTOMATIC_HEAP_SCOPE’` class in the definition of `‘Scope1’`. All the Rockall-DB library classes automatically use closest `'AUTOMATIC_HEAP_SCOPE'` of the same type (i.e. `'HEAP_TYPE'` in this case) to find the appropriate heap (i.e. `'Heap'` in this case).

Consequently, the `'AUTOMATIC_HEAP_SCOPE'` class can be used to remove the need to pass the heap (i.e. `'Heap')` as a parameter to the `'STACK'` class constructor, the call to `'PushValue'` or the class destructor. Instead, these functions contain a definition of the style shown in 'Example-06-02' below:

```
template <class HEAP,class LOCK,class VALUE>
  BOOLEAN STACK<HEAP,LOCK,VALUE>::PushValue( VALUE Value )
    {
    AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope2;

    // The rest of the implementation of the function.
    }
```

Example-06-02

The inheritance of `'Heap'` from `'Scope1'` into `'Scope2'` occurs because the type of the `'AUTOMATIC_HEAP_SCOPE'` class in `'Scope1'` matches `'Scope2'` (i.e. they are both resolve to `'SINGLE_THREADED_HEAP')`. It will be seen later that this automatic inheritance can be overridden if desired. However, in this case as no new values are provided for `'Scope2'` the values will be inherited directly from the closest instance of the same type (i.e. `'Scope1'`).

Now, the code in 'Example-06-01' only works for a `'SINGLE_THREADED_HEAP'` and a `'MULTI_THREADED_HEAP'`. A `'DATABASE_HEAP'` and a `'TRANSACTIONAL_HEAP'` both require some additional calls as shown in 'Example-06-03' below:

```
#include "AutomaticHeapScope.hpp"
#include "DatabaseHeap.hpp"
#include "RockallTypes.hpp"
#include "Stack.hpp"

typedef DATABASE_HEAP HEAP_TYPE;

int main( int Count,char *Argument[] )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope( & Heap );
  int Result = 1;

  if ( Count == 2 )
    {
    BOOLEAN Active;

    if ( (Active = (Scope.CreateFile( Argument[1] ))) )
      {
      STACK<HEAP_TYPE,NO_LOCK,int> Stack;

      if ( Stack.PushValue( 0 ) )
        { Result = 0; }
      }

    if ( Active )
      { Scope.CloseFile(); }
    }
  return Result;
  }
```

Example-06-03

The addition of the calls to `'CreateFile'` and `'CloseFile'` mean that 'Example-06-03' will now work with all of the Rockall-DB heaps.

There may seem to be a mistake in 'Example-06-03' because a `'DATABASE_HEAP'` typically requires a transaction scope for certain operations (i.e. `'New'`, `'Resize'`, `'Delete'`, etc.) and this appears to be missing. However, all of the data structure classes in the Rockall-DB library automatically create a transaction if one does not already exist, except for the `'FIXED_STRING'` and `'FLEXIBLE_STRING'` classes. Consequently, the code in 'Example-06-03' actually contains 3 transactions which are hidden within the `'STACK'` class. The first is in the constructor for the `'STACK'` class, the next is in the call to `'PushValue'` and the last is in the destructor for the `'STACK'` class. These 3 transactions can be merged into a single transaction as shown in 'Example-06-04' below:

```
#include "AutomaticHeapScope.hpp"
#include "DatabaseHeap.hpp"
#include "RockallTypes.hpp"
#include "Stack.hpp"

typedef DATABASE_HEAP HEAP_TYPE;

int main( int Count,char *Argument[] )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope( & Heap );
  int Result = 1;

  if ( Count == 2 )
    {
    BOOLEAN Active;

    if ( (Active = (Scope.CreateFile( Argument[1] ))) )
      {
      AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope2;

      if ( Scope2.BeginTransaction() )
        {
        STACK<HEAP_TYPE,NO_LOCK,int> Stack;

        if ( Stack.PushValue( 0 ) )
          { Result = 0; }
        }
      }

    if ( Active )
      { Scope.CloseFile(); }
    }

  return Result;
  }
```

Example-06-04

Now, in 'Example-06-04' we see an outer `'AUTOMATIC_HEAP_SCOPE'` called `'Scope1'` which is used to create and close the associated file. We also see an inner `'AUTOMATIC_HEAP_SCOPE'` called `'Scope2'` which is used to create a new transaction (see `'BeginTransaction'` in section 5.3.3). The `'STACK'` class will now no longer automatically create a new transaction within the `'STACK'` constructor, the call to `'PushValue'` and the `'STACK'` destructor and will instead use the existing outer transaction associated with `'Scope2'`.

Again, there may appear to be a mistake in 'Example-06-04' as there is no call to `'EndTransaction'` in the code. This is not necessary because the `'AUTOMATIC_HEAP_SCOPE'` class automatically calls `'EndTransaction'` in its destructor if a call to `'BeginTransaction'` is made without a matching call to `'EndTransaction'` (i.e. in this case `'Scope2'`). This feature of `'AUTOMATIC_HEAP_SCOPE'` is present to try to minimize the number of situations were transactions are opened but not properly closed.

Now, while it is not necessary to pass an instance of `'AUTOMATIC_HEAP_SCOPE'` as a parameter to most functions in the Rockall-DB library there is a performance impact associated with avoiding it. Consequently, an instance of `'AUTOMATIC_HEAP_SCOPE'` can optionally be passed as the first parameter to most calls in The Rockall-DB library to avoid this overhead as shown in 'Example-06-05' below:

```
#include "AutomaticHeapScope.hpp"
#include "DatabaseHeap.hpp"
#include "RockallTypes.hpp"
#include "Stack.hpp"

typedef DATABASE_HEAP HEAP_TYPE;

int main( int Count,char *Argument[] )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope( & Heap );
  int Result = 1;

  if ( Count == 2 )
    {
    BOOLEAN Active;

    if ( (Active = (Scope.CreateFile( Argument[1] ))) )
      {
      AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope2;

      if ( Scope2.BeginTransaction() )
        {
        STACK<HEAP_TYPE,NO_LOCK,int> Stack( & Scope2 );

        if ( Stack.PushValue( & Scope2,0 ) )
          { Result = 0; }
        }
      }

    if ( Active )
      { Scope.CloseFile(); }
    }
  return Result;
  }
```

Example-06-05

We have now reached the situation where ‘Example-06-05’ is almost optimal for a `'DATABASE_HEAP'`. Regardless, it should be noted that the same code will still work for a `'SINGLE_THREADED_HEAP'`, a `'MULTI_THREADED_HEAP'` and a `'TRANSACTIONAL_HEAP'` and will still be fairly optimal for all of these cases. This is because the `'AUTOMATIC_HEAP_SCOPE'` class allows the C++ compiler to automatically optimize away code that is not applicable to the associated heap.

Let’s consider the case where the `'HEAP_TYPE'` is `'SINGLE_THREADED_HEAP'` and a call is made to the `'CreateFile'` function in the `'AUTOMATIC_HEAP_SCOPE'` class. The general structure of the `'CreateFile'` function in this case is as shown in ‘Example-06-06’ below:

```
template <class HEAP>
  BOOLEAN AUTOMATIC_HEAP_SCOPE<HEAP>::CreateFile
      (
      CHAR                     *FileName,
      ENCRYPTION_FUNCTION      Function,
      VOID                     *UserValue
      )
    {
    if ( Heap -> Feature( RockallTransactionalSupport ) )
      { return (Heap -> CreateFile( FileName,Function,UserValue )); }
    else
      { return True; }
    }
```

Example-06-06

The key to understanding ‘Example-06-06’ is centered on the call to the function `'Feature'`. This function has been written so it compiles to a constant value of either `'True'` or `'False'`. When the value of the template parameter for `'HEAP'` is `'SINGLE_THREADED_HEAP'` this compiles to the constant `'False'` and after optimization by the C++ compiler the resulting code is approximately as shown in ‘Example-06-07’ below:

```
template <class HEAP>
  BOOLEAN AUTOMATIC_HEAP_SCOPE<HEAP>::CreateFile
      (
      CHAR                     *FileName,
      ENCRYPTION_FUNCTION      Function,
      VOID                     *UserValue
      )
    { return True; }
```

Example-06-07

Now, this kind of optimization occurs for any calls that are not supported by the related heap. Consequently, it is typical for almost all of the unnecessary code to simply evaporate during the optimization phase of the C++ compiler. If we take ‘Example-06-05’ above and simply change `'HEAP_TYPE'` from `'DATABASE_HEAP'` to `'SINGLE_THREADED_HEAP'` the code that actually gets generated by the C++ compiler should be close to ‘Example-06-08’ below:

```
#include "AutomaticHeapScope.hpp"
#include "SingleThreadedHeap.hpp"
#include "Stack.hpp"

typedef SINGLE_THREADED_HEAP HEAP_TYPE;

int main( int Count,char *Argument[] )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope( & Heap );
  int Result = 1;

  if ( Count == 2 )
    {
    AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope2;

    STACK<HEAP_TYPE,NO_LOCK,int> Stack( & Scope2 );

    if ( Stack.PushValue( & Scope2,0 ) )
      { Result = 0; }
    }

  return Result;
  }
```

Example-06-08

The code generated by the compiler in ‘Example-06-08’ is fairly close to the original code that we outlined at the start of this chapter in ‘Example-06-01’. Consequently, we can see that the `‘AUTOMATIC_HEAP_SCOPE’` class permits a single version of the code to be written that it works efficiently for all of the Rockall-DB heaps.

The `‘AUTOMATIC_HEAP_SCOPE’` class also supports other features to support regions (i.e. `‘CreateRegion’`, `‘ChangeRegion’` and `‘DeleteRegion’`) and strings (i.e. `‘FIXED_STRING’` and `‘FLEXIBLE_STRING’`). These features are also accessed via the constructor for the `‘AUTOMATIC_HEAP_SCOPE’` class which is approximately as shown in ‘Example-06-09’ below:

```
template <class HEAP>
  AUTOMATIC_HEAP_SCOPE<HEAP>::AUTOMATIC_HEAP_SCOPE
    (
    HEAP                          *Heap,
    SBIT32                        Region = NoRegion,
    SBIT16                        Truncate = False
    );
```

Example-06-09

We have already discussed the `‘Heap’` parameter in some detail, so we will now move on the `‘Region’` and `‘Truncate’` parameters.

When a region is created by a call to `‘CreateRegion’` it returns an `‘ID’`. A new region can be selected by calling `‘ChangeRegion’` and passing the `‘ID’` as a parameter. When the region is no longer needed `‘ChangeRegion’` can be called again but with the original `‘ID’` as a parameter. All of this can also be automated by the `‘AUTOMATIC_HEAP_SCOPE’` class by simply supplying the `‘ID’` as the second parameter to its constructor. The `‘AUTOMATIC_HEAP_SCOPE’` class will then automatically change to the required region during the scope of the instance and then automatically change back to the original region at the end of the scope.

Next, we shall take a brief look at the `'Truncate'` parameter which applies to the `'FIXED_STRING'` and `'FLEXIBLE_STRING'` classes. If the `'Truncate'` parameter is set to `'True'` then strings that are too long to be stored in an instance of either class are simply truncated instead of failing with an exception.

It can be argued that the `'AUTOMATIC_HEAP_SCOPE'` class should always be used in preference to making direct calls to the native Rockall-DB heaps. There is very little down side and it often adds a significant degree of flexibility. Regardless, the `'AUTOMATIC_HEAP_SCOPE'` class is the cornerstone of the Rockall-DB library and so a good understanding of its functionality is important to maximize its value.

6.2.2. The FixedString Class

The `'FIXED_STRING'` class supports fixed sized `'ASCII'` and `'Unicode'` strings for all of the Rockall-DB heaps. It is important to have a good understanding of the `'AUTOMATIC_HEAP_SCOPE'` class before trying to understand the `'FIXED_STRING'` and `'FLEXIBLE_STRING'` classes, as they are closely related. Any use of the `'FIXED_STRING'` and `'FLEXIBLE_STRING'` classes is entirely optional in Rockall-DB and they primarily exist to simplify the storage of strings in a `'DATABASE_HEAP'`.

The `'FIXED_STRING'` class requires three template parameters to configure it. These are the type of Rockall-DB heap, the type of characters in the string and the size of the string. The `'FIXED_STRING'` and `'FLEXIBLE_STRING'` classes are unusual in that they require a transactional scope (see `'BeginTransaction'` in section 5.3.3) when used with a `'DATABASE_HEAP'` or `'TRANSACTIONAL_HEAP'` but will *not* automatically create a new transaction if one is needed.

The `'FIXED_STRING'` and `'FLEXIBLE_STRING'` classes share much of the same code and data structures. The key difference is that a `'FIXED_STRING'` is an instance of an internal template class called `'ROCKALL::STRING'` whereas a `'FLEXIBLE_STRING'` is a pointer to an instance of this class. This may seem like a small change but the performance implications can be very significant as a `'FIXED_STRING'` typically makes no use of any related heap functionality whereas a `'FLEXIBLE_STRING'` typically makes considerable use of it.

The functions supported by the `'FIXED_STRING'` and `'FLEXIBLE_STRING'` classes are shown in 'Table-06-04' below:

Function	Description
`Assignment`	Assign a new value to a string.
`Compare`	Compare two strings.
`Join`	Join two strings.
`Lower`	Convert a string to lower case.
`Size`	Compute the number of characters in a string.
`Upper`	Convert a string to upper case.
`Value`	Supply all the characters in a string.
`'='`	See `'Assignment'` above.

`'=='` `'>'` `'>='`
`'!='` `'<'` `'<='`
	See `'Compare'` above.

`'+='`	See `'Join'` above.

Table-06-04

It is hard to cover all the functionality of the `'FIXED_STRING'` and `'FLEXIBLE_STRING'` classes without getting bogged down in details. Consequently, we shall focus on 'Example-06-10' which demonstrates most of the functionality below:

```
#include <stdio.h>

#include "AutomaticHeapScope.hpp"
#include "FixedString.hpp"
#include "RockallTypes.hpp"
#include "SingleThreadedHeap.hpp"

typedef SINGLE_THREADED_HEAP HEAP_TYPE;
typedef FIXED_STRING<HEAP_TYPE,char,64> ASCII_STRING;
typedef FIXED_STRING<HEAP_TYPE,wchar_t,64> WIDE_STRING;

int main( void )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope( & Heap );

  ASCII_STRING String1;
  ASCII_STRING String2 = "Any ascii text";
  ASCII_STRING String3 = String2;

  WIDE_STRING String4;
  WIDE_STRING String5 = L"Any wchar text";
  WIDE_STRING String6 = String5;

  FILE_SIZE StringSize;
  char Value1[64];
  wchar_t Value2[64];

  String1 = String5;
  String4 = String2;

  String1 += " + add more ascii text";
  String4 += L" + add more wchar text";

  if
      (
      (String1 == "Test ascii string")
        &&
      (String1 != String2)
        &&
      (String1 > "Test ascii string")
        &&
      (String1 >= String2)
        &&
      (String1 < "Test ascii string")
        &&
      (String1 <= String2)
        &&
      (String1.Size( & StringSize ))
        &&
      (String1.Lower())
        &&
      (String1.Size( & StringSize ))
        &&
      (String1.Upper())
        &&
      (String1.Value( & Scope,64,& StringSize,Value1 ))
      )
    { printf( "We can compare ascii strings\n" ); }

  if
      (
      (String4 == L"Test wchar string")
        &&
      (String4 != String5)
        &&
      (String4 > L"Test wchar string")
        &&
      (String4 >= String5)
        &&
      (String4 < L"Test wchar string")
        &&
      (String4 <= String5)
        &&
      (String4.Lower())
        &&
      (String4.Size( & StringSize ))
        &&
      (String4.Upper())
        &&
      (String4.Value( & Scope,64,& StringSize,Value2 ))
      )
    { printf( "We can compare wchar strings\n" ); }

  return 0;
  }
```

Example-06-10

We can see from the declarations of `‘String1’`, `‘String2’` and `‘String3’` that we can create an empty `‘ASCII’` string, assign an `‘ASCII’` string an initial value and initialize an `‘ASCII’` string from another string respectively. Furthermore, we can see from the declarations of `‘String4’`, `‘String5’` and `‘String6’` that can do the same for `‘Unicode’` strings.

We can see from the statements `‘String1 = String5’` and `‘String4 = String2’` that we can assign a `‘Unicode’` string to an `‘ASCII’` string and vice versa.

We see from the `‘if’` statements that both `‘ASCII’` and `‘Unicode’` strings can be compared to constants and other strings of the same type.

Finally, we observe a number of functions such as `‘Lower’`, `‘Size’`, `‘Upper’` and `‘Value’` which convert a string to lower case, return its current size, convert a string to upper case and return the contents respectively, for both `‘ASCII’` and `‘Unicode’` strings.

Typically, the `'FIXED_STRING'` and `'FLEXIBLE_STRING'` classes are useful in the `'HASH'`, `'QUEUE'`, `'SET'`, `'STACK'` and `'TREE'` classes to store strings within these structures.

6.2.3. The FlexibleString Class

The `'FLEXIBLE_STRING'` class supports flexible sized `'ASCII'` and `'Unicode'` strings for all of the Rockall-DB heaps. It is important to have a good understanding of the `'AUTOMATIC_HEAP_SCOPE'` class before trying to understand the `'FIXED_STRING'` and `'FLEXIBLE_STRING'` classes, as they are closely related. Any use of the `'FIXED_STRING'` and `'FLEXIBLE_STRING'` classes is entirely optional in Rockall-DB and they primarily exist to simplify the storage of strings in a `'DATABASE_HEAP`.

The `'FLEXIBLE _STRING'` class requires two template parameters to configure it. These are the type of Rockall-DB heap and the type of characters in the string. The `'FIXED_STRING'` and `'FLEXIBLE_STRING'` classes are unusual in that they require a transactional scope (see `'BeginTransaction'` in section 5.3.3) when used with a `'DATABASE_HEAP'` or `'TRANSACTIONAL_HEAP'` but will *not* automatically create a new transaction if one is needed.

The `'FIXED_STRING'` and `'FLEXIBLE_STRING'` classes share much of the same code and data structures. The key difference is that a `'FIXED_STRING'` is an instance of an internal template class called `'ROCKALL::STRING'` whereas a `'FLEXIBLE_STRING'` is a pointer to an instance of this class. This may seem like a small change but the performance implications can be very significant as a `'FIXED_STRING'` typically makes no use of any related heap functionality whereas a `'FLEXIBLE_STRING'` typically makes considerable use of it.

The functionality of the `'FIXED_STRING'` and `'FLEXIBLE_STRING'` classes is essentially identical and so the `'FLEXIBLE_STRING'` functionality is described as part of the `'FIXED_STRING'` class in 'Example-06-10' above.

6.2.4 The Hash Class

The `'HASH'` class supports variable sized hash tables (i.e. lookup tables) for all of the Rockall-DB heaps. It is important to have a good understanding of the `'AUTOMATIC_HEAP_SCOPE'` class before trying to understand the `'HASH'` class, as they are closely related.

The `'HASH'` class requires four template parameters to configure it. These are the type of Rockall-DB heap, the type of `'LOCK'`, the type of the `'KEY'` and the type of the `'VALUE'` to store. The available `'LOCK'` types are described in section 6.3 below. The `'KEY'` can be of any type, class or structure providing it supports the assignment and the comparison operators. The `'VALUE'` only needs to support the assignment operators. A simple program to create a `'HASH'` table is shown in 'Example-06-11' below:

```
#include <stdio.h>

#include "AutomaticHeapScope.hpp"
#include "FixedString.hpp"
#include "Hash.hpp"
```

```
#include "SingleThreadedHeap.hpp"

typedef SINGLE_THREADED_HEAP HEAP_TYPE;
typedef FIXED_STRING<HEAP_TYPE,char,64> KEY_TYPE;
typedef HASH<HEAP_TYPE,NO_LOCK,KEY_TYPE,int> HASH_TYPE;

int main( void )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope( & Heap );
  KEY_TYPE String = "One";
  HASH_TYPE Hash;

  if ( Hash.NewKey( String,1 ) )
    { printf( "New value stored in hash\n" ); }

  return 0;
  }
```

Example-06-11

Here we create a `'HASH'` table with `'NO_LOCK'` (i.e. see section 6.3 below) that has a `'KEY'` type of `'KEY_TYPE'` and a `'VALUE'` type of `'int'`. We then add a `'NewKey'` into the `'HASH'` with a `'KEY'` value of `'One'` and a `'VALUE'` of `'1'`.

The functions supported by the `'HASH'` class are shown in 'Table-06-05' below:

Function	Description
`DeleteAll`	Delete all of the keys in a hash table.
`DeleteKey`	Delete a key from a hash table.
`FindKey`	Find a key in a hash table and return its value.
`NewKey`	Create a new key and value pair in a hash table.
`Size`	Compute the number of keys in a hash table.
`UpdateKey`	Update the value associated with a key in a hash table.
`Walk`	Walk all the key and value pairs in a hash table.

Table-06-05

An extended example including all of the functions in 'Table-06-05' is shown in 'Example-06-12' below:

```
#include <stdio.h>

#include "AutomaticHeapScope.hpp"
#include "FixedString.hpp"
#include "Hash.hpp"
#include "RockallTypes.hpp"
#include "SingleThreadedHeap.hpp"

typedef SINGLE_THREADED_HEAP HEAP_TYPE;
typedef FIXED_STRING<HEAP_TYPE,char,64> KEY_TYPE;
typedef HASH<HEAP_TYPE,NO_LOCK,KEY_TYPE,int> HASH_TYPE;

static BOOLEAN WalkFunction( KEY_TYPE & Key,int & Value,void *User )
  {
  int *UserValue = ((int*) User);

  printf( "Value: %d, User Value: %d\n",Value,(*UserValue) );

  return True;
  }

int main( void )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope( & Heap );
  KEY_TYPE String = "One";
  HASH_TYPE Hash;
  FILE_SIZE Size;
  int Value;

  if
      (
      (Hash.NewKey( String,1,False ))
        &&
      (Hash.FindKey( String,& Value ))
        &&
      (Hash.Size( & Size ))
        &&
      (Hash.UpdateKey( String,2 ))
        &&
      (Hash.Walk( WalkFunction,((void*) & Value) ))
        &&
      (Hash.DeleteKey( String ))
        &&
      (Hash.DeleteAll())
      )
    { printf( "End of example\n" ); }

  return 0;
  }
```

Example-06-12

The call to `'NewKey'` adds a new `'KEY'` and `'VALUE'` pair into the `'Hash'` table. The `'KEY'` in this case will be `'One'` and the `'VALUE'` will be `'1'`. A Rockall-DB `'HASH'` table does not support duplicate `'KEY'` values and the final parameter `'False'` causes the call to fail if the new `'KEY'` is a duplicate. Alternately, if the final parameter is set to `'True'` then the new `'VALUE'` would be used to overwrite any existing value.

The call to `'FindKey'` looks up a `'KEY'` in a `'Hash'` table and returns the associated value. The `'KEY'` in this case will be `'One'` and the `'VALUE'` returned will be `'1'`.

The call to `'Size'` returns the number of `'KEY'` values in the `'Hash'` table. The size of the `'Hash'` table in this case is `'1'`.

The call to `'UpdateKey'` looks up a `'KEY'` in a `'Hash'` table and updates the associated value. The `'KEY'` in this case will be `'One'` and the `'VALUE'` will be updated to `'2'`.

The call to `'Walk'` leads to the `'WalkFunction'` being called once for every entry in the `'Hash'` table. The `'WalkFunction'` is called with the each `'Key'` and `'Value'` pair along with the second parameter passed to the original `'Walk'` function call (i.e. `'& Value'` in this case). The `'WalkFunction'` returns `'True'` if it wants to continue to the next `'Key'` and `'Value'` pair and `'False'` otherwise.

The call to `'DeleteKey'` deletes a `'KEY'` from a `'Hash'` table. The `'KEY'` in this case will be `'One'`.

The call to `'DeleteAll'` deletes all the `'KEY'` values from a `'Hash'` table.

All of the functions in the `'HASH'` class may optionally be called with a pointer to an `'AUTOMATIC_HEAP_SCOPE'` as the first parameter to improve their performance. Consequently, 'Example-06-12' could be rewritten as shown in 'Example-06-13' below:

```
#include <stdio.h>

#include "AutomaticHeapScope.hpp"
#include "FixedString.hpp"
#include "Hash.hpp"
#include "RockallTypes.hpp"
#include "SingleThreadedHeap.hpp"

typedef SINGLE_THREADED_HEAP HEAP_TYPE;
typedef FIXED_STRING<HEAP_TYPE,char,64> KEY_TYPE;
typedef HASH<HEAP_TYPE,NO_LOCK,KEY_TYPE,int> HASH_TYPE;

static BOOLEAN WalkFunction( KEY_TYPE & Key,int & Value,void *User )
  {
  int *UserValue = ((int*) User);

  printf( "Value: %d, User Value: %d\n",Value,(*UserValue) );

  return True;
  }

int main( void )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope( & Heap );
  KEY_TYPE String = "One";
  HASH_TYPE Hash( & Scope );
  FILE_SIZE Size;
  int Value;

  if
      (
      (Hash.NewKey( & Scope,String,1,False ))
        &&
      (Hash.FindKey( & Scope,String,& Value ))
        &&
      (Hash.Size( & Scope,& Size ))
        &&
      (Hash.UpdateKey( & Scope,String,2 ))
        &&
      (Hash.Walk( & Scope,WalkFunction,((void*) & Value) ))
        &&
      (Hash.DeleteKey( & Scope,String ))
        &&
      (Hash.DeleteAll( & Scope ))
      )
    { printf( "End of example\n" ); }
  return 0;
  }
```

Example-06-13

6.2.5 The Queue Class

The `'QUEUE'` class supports variable sized 'First in First Out' tables (i.e. FIFO table) for all of the Rockall-DB heaps. It is important to have a good understanding of the `'AUTOMATIC_HEAP_SCOPE'` class before trying to understand the `'QUEUE'` class, as they are closely related.

The `'QUEUE'` class requires three template parameters to configure it. These are the type of Rockall-DB heap, the type of `'LOCK'` and the type of the `'VALUE'` to store. The available `'LOCK'` types are described in section 6.3 below. The `'VALUE'` can be of any type, class or structure providing it supports the assignment operators. A simple program to create a `'QUEUE'` table is shown in 'Example-06-14' below:

```
#include "AutomaticHeapScope.hpp"
#include "Queue.hpp"
#include "SingleThreadedHeap.hpp"

typedef SINGLE_THREADED_HEAP HEAP_TYPE;
typedef QUEUE<HEAP_TYPE,NO_LOCK,int> QUEUE_TYPE;

int main( void )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope( & Heap );
  QUEUE_TYPE Queue;

  if ( Queue.PushValue( 1 ) )
    { printf( "New value stored in queue\n" ); }

  return 0;
  }
```

Example-06-14

Here we create a `'QUEUE'` with `'NO_LOCK'` (i.e. see section 6.3 below) and a `'VALUE'` type of `'int'` and then `'PushValue'` a `'VALUE'` of `'1'` into the `'QUEUE'`.

The functions supported by the `'QUEUE'` class are shown in 'Table-06-06' below:

Function	Description
`DeleteAll`	Delete all of the values in a queue.
`PeakValue`	Examine the first value in a queue.
`PopValue`	Pop the first value from a queue.
`PushValue`	Push a new value into a queue.
`Size`	Compute the number of values in a queue.
`Walk`	Walk all the values in a queue.

Table-06-06

An extended example including all of the functions in 'Table-06-06' is shown in 'Example-06-15' below:

```
#include <stdio.h>
```

```
#include "AutomaticHeapScope.hpp"
#include "Queue.hpp"
#include "RockallTypes.hpp"
#include "SingleThreadedHeap.hpp"

typedef SINGLE_THREADED_HEAP HEAP_TYPE;
typedef QUEUE<HEAP_TYPE,NO_LOCK,int> QUEUE_TYPE;

static BOOLEAN WalkFunction( int & Value,void *User )
  {
  int *UserValue = ((int*) User);

  printf( "Value: %d, User Value: %d\n",Value,(*UserValue) );

  return True;
  }

int main( void )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope( & Heap );
  QUEUE_TYPE Queue;
  FILE_SIZE Size;
  int Value;

  if
      (
      (Queue.PushValue( 1 ))
        &&
      (Queue.PeekValue( & Value ))
        &&
      (Queue.Size( & Size ))
        &&
      (Queue.Walk( WalkFunction,((void*) & Value) ))
        &&
      (Queue.PopValue( & Value ))
        &&
      (Queue.DeleteAll())
      )
    { printf( "End of example\n" ); }

  return 0;
  }
```

Example-06-15

The call to `‘PushValue’` pushes a `‘VALUE’` into the `‘Queue’`. The `‘VALUE’` in this case will be `‘1’`.

The call to `‘PeekValue’` returns the first value in the `‘Queue’` without removing it. The `‘VALUE’` in this case will be `‘1’`.

The call to `‘Size’` returns the number of values in the `‘Queue’`. The size of the `‘Queue’` in this case will be `‘1’`.

The call to `‘Walk’` leads to the `‘WalkFunction’` being called once for every entry in the `‘Queue’`. The `‘WalkFunction’` is called with the each `‘Value’` along with the second parameter passed to the original `‘Walk’` function call (i.e. `‘& Value’` in this case). The `‘WalkFunction’` returns `‘True’` if it wants to continue to the next `‘Value’` and `‘False’` otherwise.

The call to `‘PopValue’` returns the first value in the `‘Queue’` and removes it from the `‘Queue’`. The `‘VALUE’` in this case will be `‘1’`.

The call to `‘DeleteAll’` deletes all the values in the `‘Queue’`.

All of the functions in the `‘QUEUE’` class may optionally be called with a pointer to an `‘AUTOMATIC_HEAP_SCOPE’` as the first parameter to improve their performance. Consequently, ‘Example-06-15’ could be rewritten as shown in ‘Example-06-16’ below:

```
#include <stdio.h>

#include "AutomaticHeapScope.hpp"
#include "Queue.hpp"
#include "RockallTypes.hpp"
#include "SingleThreadedHeap.hpp"

typedef SINGLE_THREADED_HEAP HEAP_TYPE;
typedef QUEUE<HEAP_TYPE,NO_LOCK,int> QUEUE_TYPE;

static BOOLEAN WalkFunction( int & Value,void *User )
  {
  int *UserValue = ((int*) User);

  printf( "Value: %d, User Value: %d\n",Value,(*UserValue) );

  return True;
  }

int main( void )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope( & Heap );
  QUEUE_TYPE Queue( & Scope );
  FILE_SIZE Size;
  int Value;

  if
      (
      (Queue.PushValue( & Scope,1 ))
        &&
      (Queue.PeekValue( & Scope,& Value ))
        &&
      (Queue.Size( & Scope,& Size ))
        &&
      (Queue.Walk( & Scope,WalkFunction,((void*) & Value) ))
        &&
      (Queue.PopValue( & Scope,& Value ))
        &&
      (Queue.DeleteAll( & Scope ))
      )
    { printf( "End of example\n" ); }

  return 0;
  }
```

Example-06-16

6.2.6 The RowSet Class

The `'ROW_SET'` class supports variable sized sorted collection of `'FILE_ADDRESS'` values for all of the Rockall-DB heaps. It is important to have a good understanding of the `'AUTOMATIC_HEAP_SCOPE'` class before trying to understand the `'ROW_SET'` class, as they are closely related.

The `'ROW_SET'` class requires two template parameters to configure it. These are the type of Rockall-DB heap and the type of `'LOCK'`. The available `'LOCK'` types are described in section 6.3 below.

The `'ROW_SET'` class is directly inherited from the `'SET'` class (see the `'SET'` class in section 6.2.7 below) as shown in 'Example-06-17' below:

```
template <class HEAP,class LOCK> class ROW_SET :
  public SET<HEAP,LOCK,FILE_ADDRESS>
```

Example-06-17

Consequently, all of the `'ROW_SET'` class functionality is directly derived from the `'SET'` class (i.e. see the `'SET'` class for details) except for the `'Touch'` function. The `'Touch'` function simply ensures that every `'FILE_ADDRESS'` in a `'ROW_SET'` is asynchronously bought into main memory. An example of the `'ROW_SET'` class and the `'Touch'` function is shown in 'Example-06-18' below:

```
#include <stdio.h>

#include "AutomaticHeapScope.hpp"
#include "RowSet.hpp"
#include "SingleThreadedHeap.hpp"

typedef SINGLE_THREADED_HEAP HEAP_TYPE;
typedef ROW_SET<HEAP_TYPE,NO_LOCK> SET_TYPE;

int main( void )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope( & Heap );
  SET_TYPE RowSet( & Scope );

  if ( RowSet.Touch( & Scope ) )
    { printf( "End of example\n" ); }

  return 0;
  }
```

Example-06-18

The parameter value of `'& Scope'` in the call to `'Touch'` in 'Example-06-18' is optional and may be omitted in the same way as in `'HASH'` and `'QUEUE'` classes above. Additionally, in this particular example the `'RowSet'` is empty and so the call to `'Touch'` would have no effect, as there will be no `'FILE_ADDRESS'` values to be asynchronously bought into memory.

6.2.7 The Set Class

The `'SET'` class supports variable sized sorted collections of values for all of the Rockall-DB heaps. It is important to have a good understanding of the `'AUTOMATIC_HEAP_SCOPE'` class before trying to understand the `'SET'` class, as they are closely related.

The `'SET'` class requires three template parameters to configure it. These are the type of Rockall-DB heap, the type of `'LOCK'` and the type of the `'VALUE'` to store. The available `'LOCK'` types are described in section 6.3 below. The `'VALUE'` can be of any type, class or structure providing it supports the assignment operators. A simple program to create a `'SET'` is shown in 'Example-06-19' below:

```
#include <stdio.h>

#include "AutomaticHeapScope.hpp"
#include "Set.hpp"
#include "SingleThreadedHeap.hpp"

typedef SINGLE_THREADED_HEAP HEAP_TYPE;
typedef SET<HEAP_TYPE,NO_LOCK,int> SET_TYPE;

int main( void )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope( & Heap );
  SET_TYPE Set;

  if ( Set.NewValue( 1 ) )
    { printf( "New value stored in set\n" ); }

  return 0;
  }
```

Example-06-19

Here we create a `'SET'` with `'NO_LOCK'` (i.e. see section 6.3 below) and a `'VALUE'` type of `'int'` and then add a `'NewValue'` of `'1'` in the `'SET'`.

The functions supported by the `'SET'` class are shown in 'Table-06-07' below:

Function	Description
`Copy`	Delete the current set and copy in the contents of another set.
`DeleteAll`	Delete all of the values in a set.
`DeleteValue`	Delete a value in a set.
`Difference`	Keep only the values that are *not* in both sets.
`FindValue`	Find a value in a set.
`Intersect`	Keep only the values that are in both sets.
`Join`	Keep all the values from both sets.
`NewValue`	Create a new value in a set.
`Size`	Compute the number of values in a set.
`Value`	Supply all the values in a set.
`Walk`	Walk all the values in a set.

Table-06-07

An extended example including all of the functions in 'Table-06-07' is shown in 'Example-06-20' below:

```
#include <stdio.h>

#include "AutomaticHeapScope.hpp"
#include "RockallTypes.hpp"
#include "Set.hpp"
#include "SingleThreadedHeap.hpp"

typedef SINGLE_THREADED_HEAP HEAP_TYPE;
typedef SET<HEAP_TYPE,NO_LOCK,int> SET_TYPE;

static BOOLEAN WalkFunction( int & Value,void *User )
  {
  int *UserValue = ((int*) User);

  printf( "Value: %d, User Value: %d\n",Value,(*UserValue) );

  return True;
  }

int main( void )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope( & Heap );
  SET_TYPE Set1;
  SET_TYPE Set2;
  SET_TYPE Set3;
  FILE_SIZE Size;
  int Values[64];

  if
      (
      (Set1.NewValue( 1 ))
        &&
      (Set2.Copy( & Set1 ))
        &&
      (Set3.Difference( & Set1 ))
        &&
      (Set1.FindValue( 1 ))
        &&
      (Set2.Intersect( & Set1 ))
        &&
      (Set2.Join( & Set1 ))
        &&
      (Set1.Size( & Size ))
        &&
      (Set1.Value( 64,& Size,Values ))
        &&
      (Set1.Walk( WalkFunction,((void*) & Values[0]) ))
        &&
      (Set1.DeleteValue( 1 ))
        &&
      (Set1.DeleteAll())
      )
    { printf( "End of example\n" ); }

  return 0;
  }
```

Example-06-20

The call to `'NewValue'` adds a `'VALUE'` into a set. The set in this case is `'Set1'` and the `'VALUE'` is `'1'`.

The call to `'Copy'` copies the contents of one set to another. The destination set in this case is `'Set2'` and source set is `'Set1'`.

The call to `'Difference'` computes the difference between two sets. The destination set in this case is `'Set3'` and source set is `'Set1'`. We can see that the destination `'Set3'` will be empty and so in the difference will be all the values found in `'Set1'`.

The call to `'FindValue'` will return `'True'` if the value is in a set and `'False'` otherwise. The set in this case is `'Set1'` and *does* contain the value `'1'`, so the function will return `'True'`.

The call to `'Intersect'` computes the intersection of two sets. The destination set in this case is `'Set2'` and source set is `'Set1'`. We can see that the destination `'Set2'` is a `'Copy'` of `'Set1'` and so all the values from both sets will be in the result set.

The call to `'Join'` computes the union of two sets. The destination set in this case is `'Set2'` and source set is `'Set1'`. We can see that the destination `'Set2'` is a `'Copy'` of `'Set1'` and so after removing duplicates all the values from both sets will be in the result set.

The call to `'Size'` returns the number of values in the set. The set in this case is `'Set1'` and its size is `'1'`.

The call to `'Value'` returns the all of the values in the set. The set in this case is `'Set1'` and its size is `'1'`. Consequently, as the `'MaxSize'` of `'64'` exceeds the actual size of `'Set1'` the `'Size'` parameter will be set to `'1'` and the contents of `'Values'` will be set to the associated set values.

The call to `'Walk'` leads to the `'WalkFunction'` being called once for every entry in the set. The `'WalkFunction'` is called with the each `'Value'` along with the second parameter passed to the original `'Walk'` function call (i.e. `'& Values[0]'` in this case). The `'WalkFunction'` returns `'True'` if it wants to continue to the next `'Value'` and `'False'` otherwise.

The call to `'DeleteValue'` deletes a value in a set. The set in this case is `'Set1'` and the value is `'1'`.

The call to `'DeleteAll'` deletes all the values in a set.

All of the functions in the `'SET'` class may optionally be called with a pointer to an `'AUTOMATIC_HEAP_SCOPE'` as the first parameter to improve their performance. Consequently, 'Example-06-20' could be rewritten as shown in 'Example-06-21' below:

```
#include <stdio.h>

#include "AutomaticHeapScope.hpp"
```

```
#include "RockallTypes.hpp"
#include "Set.hpp"
#include "SingleThreadedHeap.hpp"

typedef SINGLE_THREADED_HEAP HEAP_TYPE;
typedef SET<HEAP_TYPE,NO_LOCK,int> SET_TYPE;

static BOOLEAN WalkFunction( int & Value,void *User )
  {
  int *UserValue = ((int*) User);

  printf( "Value: %d, User Value: %d\n",Value,(*UserValue) );

  return True;
  }

int main( void )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope( & Heap );
  SET_TYPE Set1( & Scope );
  SET_TYPE Set2( & Scope );
  SET_TYPE Set3( & Scope );
  FILE_SIZE Size;
  int Values[64];

  if
      (
      (Set1.NewValue( & Scope,1 ))
        &&
      (Set2.Copy( & Scope,& Set1 ))
        &&
      (Set3.Difference( & Scope,& Set1 ))
        &&
      (Set1.FindValue( & Scope,1 ))
        &&
      (Set2.Intersect( & Scope,& Set1 ))
        &&
      (Set2.Join( & Scope,& Set1 ))
        &&
      (Set1.Size( & Scope,& Size ))
        &&
      (Set1.Value( & Scope,64,& Size,Values ))
        &&
      (Set1.Walk( & Scope,WalkFunction,((void*) & Values[0]) ))
        &&
      (Set1.DeleteValue( & Scope,1 ))
        &&
      (Set1.DeleteAll( & Scope ))
      )
    { printf( "End of example\n" ); }

  return 0;
  }
```

Example-06-21

6.2.8 The Stack Class

The ‘`STACK`’ class supports variable sized ‘Last in First Out’ tables (i.e. LIFO table) for all of the Rockall-DB heaps. It is important to have a good understanding of the

`'AUTOMATIC_HEAP_SCOPE'` class before trying to understand the `'STACK'` class, as they are closely related.

The `'STACK'` class requires three template parameters to configure it. These are the type of Rockall-DB heap, the type of `'LOCK'` and the type of the `'VALUE'` to store. The available `'LOCK'` types are described in section 6.3 below. The `'VALUE'` can be of any type, class or structure providing it supports the assignment operators.

The `'QUEUE'` and `'STACK'` classes support the same functional interfaces and operate in similar ways, except for being `'QUEUE'` and `'STACK'` structures respectively. Consequently, see the `'QUEUE'` class in section 6.2.5 for further details and related examples for the `'STACK'` class.

6.2.9 The Tree Class

The `'TREE'` class supports configurable, variable sized, multi-way trees (i.e. an index) for all of the Rockall-DB heaps. It is important to have a good understanding of the `'AUTOMATIC_HEAP_SCOPE'` class before trying to understand the `'TREE'` class, as they are closely related.

The `'TREE'` class requires four mandatory template parameters to configure it and also supports an additional two optional template parameters. The mandatory template parameters are the type of Rockall-DB heap, the type of `'LOCK'`, the type of the `'KEY'` and the type of the `'VALUE'` to store. The available `'LOCK'` types are described in section 6.3 below. The `'KEY'` can be of any type, class or structure providing it supports the assignment and the comparison operators. The `'VALUE'` only needs to support the assignment operators.

The optional template parameters are the number of `'VALUE'` nodes in a twig (i.e. leaves on a twig) and the number of stems in a branch (i.e. twigs on a branch). The value of these template parameters must be in the range 16 to 16,384 and control the overall shape of the `'TREE'`. A smaller value will tend to save space and reduce contention while increasing the number of levels in the `'TREE'`. A larger value will tend to consume more space and increase contention but reduce the number of levels in the `'TREE'` (i.e. a tree with 268,435,456 entries will create 7 levels of indexes with the minimum values of 16 whereas with the maximum values of 16,384 it will only create only 1 level of index). A simple program to create a `'TREE'` table is shown in 'Example-06-22' below:

```
#include <stdio.h>

#include "AutomaticHeapScope.hpp"
#include "FixedString.hpp"
#include "SingleThreadedHeap.hpp"
#include "Tree.hpp"

typedef SINGLE_THREADED_HEAP HEAP_TYPE;
typedef FIXED_STRING<HEAP_TYPE,char,64> KEY_TYPE;
typedef TREE<HEAP_TYPE,NO_LOCK,KEY_TYPE,int> TREE_TYPE;

int main( void )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope( & Heap );
  KEY_TYPE String = "One";
  TREE_TYPE Tree;

  if ( Tree.NewKey( String,1 ) )
    { printf( "New value stored in tree\n" ); }

  return 0;
  }
```

Example-06-22

Here we create a `'TREE'` with `'NO_LOCK'` (i.e. see section 6.3 below) that has a `'KEY'` type of `'KEY_TYPE'` and a `'VALUE'` type of `'int'`. We then add a `'NewKey'` into the `'TREE'` with a `'KEY'` value of `'One'` and a `'VALUE'` of `'1'`.

The functions supported by the `'TREE'` class are shown in 'Table-06-08' below:

Function	Description
`DeleteAll`	Delete all of the keys in a tree.
`DeleteKey`	Delete a key from a tree.
`FindKey`	Find a key in a tree and return its value.
`FindKeys`	Find a range of key and value pairs in a tree.
`NewKey`	Create a new key and value pair in a tree.
`Size`	Compute the number of keys in a tree.
`UpdateKey`	Update the value associated with a key in a tree.
`Walk`	Walk all the key and value pairs in a tree.

Table-06-08

An extended example including all of the functions in 'Table-06-08' is shown in 'Example-06-23' below:

```
#include <stdio.h>

#include "AutomaticHeapScope.hpp"
#include "FixedString.hpp"
#include "RockallTypes.hpp"
#include "SingleThreadedHeap.hpp"
#include "Tree.hpp"

typedef SINGLE_THREADED_HEAP HEAP_TYPE;
typedef FIXED_STRING<HEAP_TYPE,char,64> KEY_TYPE;
typedef TREE<HEAP_TYPE,NO_LOCK,KEY_TYPE,int> TREE_TYPE;

static BOOLEAN WalkFunction( KEY_TYPE & Key,int & Value,void *User )
  {
  int *UserValue = ((int*) User);

  printf( "Value: %d, User Value: %d\n",Value,(*UserValue) );

  return True;
  }

int main( void )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope( & Heap );
  KEY_TYPE String1 = "One";
  KEY_TYPE String2 = "A";
  KEY_TYPE String3 = "Z";
  KEY_TYPE String4;
  TREE_TYPE Tree;
  FILE_SIZE Size;
  int Value;

  if
      (
      (Tree.NewKey( String1,1,False ))
        &&
      (Tree.FindKey( String1,& Value ))
        &&
      (Tree.FindKeys( String2,String3,1,& Size,& String4,& Value ))
        &&
      (Tree.Size( & Size ))
        &&
      (Tree.UpdateKey( String1,2 ))
        &&
      (Tree.Walk( WalkFunction,((void*) & Value) ))
        &&
      (Tree.DeleteKey( String1 ))
        &&
      (Tree.DeleteAll())
      )
    { printf( "End of example\n" ); }

  return 0;
  }
```

Example-06-23

The call to `'NewKey'` adds a new `'KEY'` and `'VALUE'` pair into the `'Tree'`. The `'KEY'` in this case will be `'One'` and the `'VALUE'` will be `'1'`. A Rockall-DB `'TREE'` does not support duplicate `'KEY'` values and the final parameter `'False'` causes the call to fail if the new `'KEY'` is a duplicate. Alternately, if the final parameter is set to `'True'` then the new `'VALUE'` would be used to overwrite any existing value.

The call to `'FindKey'` looks up a `'KEY'` in a `'Tree'` and returns the associated value. The `'KEY'` in this case will be `'One'` and the `'VALUE'` returned will be `'1'`.

The call to `'FindKeys'` returns a range of keys and associated values in a `'Tree'`. In this case, the lower bound would be `'A'` and the higher bound would be `'Z'`. The maximum number of values returned will be `'1'`. The number of values returned will be supplied in `'Size'` and the actual keys and values matched will be returned in `'String4'` and `'Value'`. If the maximum number of values was greater than `'1'` then `'String4'` and `'Value'` would need to be arrays of the same size and the results would be supplied in sorted order.

The call to `'Size'` returns the number of `'KEY'` values in a `'Tree'`. The size of the `'Tree'` in this case is `'1'`.

The call to `'UpdateKey'` looks up a `'KEY'` in a `'Tree'` and updates the associated value. The `'KEY'` in this case will be `'One'` and the updated `'VALUE'` will be `'2'`.

The call to `'Walk'` leads to the `'WalkFunction'` being called once for every entry in the `'Tree'`. The `'WalkFunction'` is called with the each `'Key'` and `'Value'` pair in sorted order along with the second parameter passed to the original `'Walk'` function call (i.e. `'& Value'` in this case). The `'WalkFunction'` returns `'True'` if it wants to continue to the next `'Key'` and `'Value'` pair and `'False'` otherwise.

The call to `'DeleteKey'` deletes a `'KEY'` from a `'Tree'`. The `'KEY'` in this case would be `'One'`.

The call to `'DeleteAll'` deletes all the `'KEY'` values from a `'Tree'`.

All of the functions in the `'TREE'` class may optionally be called with a pointer to an `'AUTOMATIC_HEAP_SCOPE'` as the first parameter to improve their performance. Consequently, 'Example-06-23' could be rewritten as shown in 'Example-06-24' below:

```
#include <stdio.h>

#include "AutomaticHeapScope.hpp"
#include "FixedString.hpp"
#include "RockallTypes.hpp"
#include "SingleThreadedHeap.hpp"
#include "Tree.hpp"

typedef SINGLE_THREADED_HEAP HEAP_TYPE;
typedef FIXED_STRING<HEAP_TYPE,char,64> KEY_TYPE;
typedef TREE<HEAP_TYPE,NO_LOCK,KEY_TYPE,int> TREE_TYPE;

static BOOLEAN WalkFunction( KEY_TYPE & Key,int & Value,void *User )
  {
  int *UserValue = ((int*) User);

  printf( "Value: %d, User Value: %d\n",Value,(*UserValue) );

  return True;
  }

int main( void )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope( & Heap );
  KEY_TYPE String1 = "One";
  KEY_TYPE String2 = "A";
  KEY_TYPE String3 = "Z";
  KEY_TYPE String4;
  TREE_TYPE Tree( & Scope );
  FILE_SIZE Size;
  int Value;

  if
      (
      (Tree.NewKey( & Scope,String1,1,False ))
        &&
      (Tree.FindKey( & Scope,String1,& Value ))
        &&
      (Tree.FindKeys( & Scope,String2,String3,1,& Size,& String4 ))
        &&
      (Tree.Size( & Scope,& Size ))
        &&
      (Tree.UpdateKey( & Scope,String1,2 ))
        &&
      (Tree.Walk( & Scope,WalkFunction,((void*) & Value) ))
        &&
      (Tree.DeleteKey( & Scope,String1 ))
        &&
      (Tree.DeleteAll( & Scope ))
      )
    { printf( "End of example\n" ); }

  return 0;
  }
```

Example-06-24

6.3. The Rockall-DB Library Lock Classes

A significant portion of the Rockall-DB library outlined in section 6.2 above requires a `'LOCK'` to be supplied as a template parameter. A number of pre-built `'LOCK'` types are available in Rockall-DB which may be used in these classes or independently. All of the supported `'LOCK'` types are listed in 'Table-06-09' below:

File Name	Sections	Description
AutomaticLockScope.hpp	6.3.1.	A class to automate lock management.
FileLock.hpp	6.3.2.	A lock suitable for file or memory structures.
MemoryLock.hpp	6.3.3.	A lock suitable for memory structures.
NoLock.hpp	6.3.4.	A class to optimize away locking code.
TransactionLock.hpp	6.3.5.	A lock related to transactions.

Table-06-09

All of the Rockall-DB library locks support the same interface, even though their implementations often vary significantly. Consequently, any Rockall-DB `'LOCK'` can trivially be substituted for a different `'LOCK'` type. The overall structure of the interface is shown in 'Example-06-25' below:

```
class FILE_LOCK
  {
  public:
    FILE_LOCK( VOID );

    BOOLEAN ClaimExclusiveLock( BOOLEAN Wait = True );

    BOOLEAN ClaimSharedLock( BOOLEAN Wait = True );

    VOID ReleaseExclusiveLock( VOID );

    VOID ReleaseSharedLock( VOID );

    ~FILE_LOCK( VOID );
  };
```

Example-06-25

A call to `'ClaimExclusiveLock'` with `'Wait'` set to `'True'` will not return from the call until the current thread is the sole owner of the associated lock and will always return the value `'True'`. If `'Wait'` set to `'False'` the call will return immediately and the returned value will be `'True'` if the lock was claimed or `'False'` otherwise.

A call to `'ClaimSharedLock'` with `'Wait'` set to `'True'` will not return from the call until the current thread is a shared owner of the associated lock and will always return the value `'True'`. If `'Wait'` set to `'False'` the call will return immediately and the returned value will be `'True'` if the shared lock was claimed or `'False'` otherwise.

A call to `'ReleaseExclusiveLock'` or `'ReleaseSharedLock'` are the converse of a call to `'ClaimExclusiveLock'` or `'ClaimSharedLock'` respectively.

The Rockall-DB library does not support mismatching or recursive lock calls. Consequently, any attempt to call both `'ClaimExclusiveLock'` and `'ClaimSharedLock'` or to make multiple calls to `'ClaimExclusiveLock'` without prior matching calls to `'ReleaseExclusiveLock'` or `'ReleaseSharedLock'` respectively will result in deadlock. Any attempt to call `'ReleaseExclusiveLock'` or `'ReleaseSharedLock'` without a prior matching call to `'ClaimExclusiveLock'` or `'ClaimSharedLock'` respectively will result in a software failure.

There is no requirement to use the Rockall-DB library locks classes if suitable alternatives are available (i.e. alternatives that support the interface outlined above).

6.3.1. The AutomaticLockScope Class

The `'AUTOMATIC_LOCK_SCOPE'` class automatically claims a Rockall-DB `'LOCK'` when a block of code is entered and automatically releases it when the block is exited. There is no requirement to use `'AUTOMATIC_LOCK_SCOPE'` class in Rockall-DB and it is provided purely for convenience. An example of `'AUTOMATIC_LOCK_SCOPE'` class is shown in 'Example-06-26' below:

```
#include <stdio.h>

#include "AutomaticLockScope.hpp"
#include "FileLock.hpp"

int main( void )
  {
  FILE_LOCK Lock;

  // No lock owned.
  Lock.ClaimExclusiveLock();
  // Exclusive lock owned.
  Lock.ReleaseExclusiveLock();

  // No lock owned.
  {
  AUTOMATIC_EXCLUSIVE_LOCK_SCOPE<FILE_LOCK> ExclusiveLock( & Lock );
  // Exclusive lock owned.
  }

  // No lock owned.
  {
  AUTOMATIC_EXCLUSIVE_LOCK_SCOPE<FILE_LOCK> ExclusiveLock( & Lock );
  // Exclusive lock owned.
  ExclusiveLock.Release();
  // No lock owned.
  ExclusiveLock.Reclaim();
  // Exclusive lock owned.
  }

  // No lock owned.
  Lock.ClaimSharedLock();
  // Shared lock owned.
  Lock.ReleaseSharedLock();

  // No lock owned.
  {
  AUTOMATIC_SHAREABLE_LOCK_SCOPE<FILE_LOCK> SharedLock( & Lock );
  // Shared lock owned.
  }

  // No lock owned.
  return 0;
  }
```

Example-06-26

We see at the beginning of 'Example-06-26' a `'FILE_LOCK'` called `'Lock'`. We then see that this `'Lock'` can be manually claimed by calls such as `'Lock.ClaimExclusiveLock()'` and `'Lock.ClaimSharedLock()'` and manually released by related calls such as `'Lock.ReleaseExclusiveLock()'` and `'Lock.ReleaseSharedLock()'` respectively.

We also see how the `'AUTOMATIC_EXCLUSIVE_LOCK_SCOPE'` class and the `'AUTOMATIC_SHAREABLE_LOCK_SCOPE'` class can be used to automate these exclusive lock and shared lock calls within a block respectively.

Finally, we see that the `'Release'` function can optionally be used to release lock early and the `'Reclaim'` function can be used to optionally reclaim the lock again (i.e. if it was released earlier).

6.3.2. The FileLock Class

A `'FILE_LOCK'` is suitable for all Rockall-DB heaps and Rockall-DB library classes. The general structure and patterns of use of a `'FILE_LOCK'` are outlined in sections 6.3 and 6.3.1 above.

A `'FILE_LOCK'` consists of a single byte of memory which remains unchanged. However, the address of this byte of memory is used to find an associated `'MEMORY_LOCK'` (see section 6.3.3 below) elsewhere in the process space which implements the actual locking calls. A `'FILE_LOCK'` is the lowest performing of the Rockall-DB library lock classes but is important as *only* `'FILE_LOCK'` and `'NO_LOCK'` may be used in conjunction with a `'DATABASE_HEAP'`.

6.3.3. The MemoryLock Class

A `'MEMORY_LOCK'` is suitable for all Rockall-DB heaps and Rockall-DB library classes except when they are used in connection with a `'DATABASE_HEAP'`. The general structure and patterns of use of a `'MEMORY_LOCK'` are outlined in sections 6.3 and 6.3.1 above.

A `'MEMORY_LOCK'` is a high performance spinlock that is optimized for multi-threaded applications. Typically, a `'MEMORY_LOCK'` should be used in preference to a `'FILE_LOCK'` for performance reasons, except for where an object is stored in a file (i.e. a `'DATABASE_HEAP'`). If a `'MEMORY_LOCK'` is stored in a file (i.e. a `'DATABASE_HEAP'`) the code is almost guaranteed to fail when the object is reloaded into main memory in a new process.

6.3.4. The NoLock Class

The `'NO_LOCK'` class is suitable for all Rockall-DB heaps and Rockall-DB library classes. The general structure and patterns of use of a `'NO_LOCK'` are outlined in sections 6.3 and 6.3.1 above.

An instance of `'NO_LOCK'` simply helps the C++ compiler to optimize away all the related locking code. The structure of The `'NO_LOCK'` class is shown in 'Example-06-27' below:

```
class NO_LOCK
  {
  public:
    NO_LOCK( VOID )
      { /* void */ }

    BOOLEAN ClaimExclusiveLock( BOOLEAN Wait = True )
      { return True; }

    BOOLEAN ClaimSharedLock( BOOLEAN Wait = True )
      { return True; }

    VOID ReleaseExclusiveLock( VOID )
      { /* void */ }

    VOID ReleaseSharedLock( VOID )
      { /* void */ }

    ~NO_LOCK( VOID )
      { /* void */ }
  };
```

Example-06-27

We can see from 'Example-06-27' that the `'NO_LOCK'` class has no functional code. Therefore, even the most intellectually challenged C++ compiler should be able to easily optimize it away entirely. Consequently, using `'NO_LOCK'` with the Rockall-DB library classes automatically removes all of the related locking code.

6.3.5. The TransactionLock Class

A `'TRANSACTION_LOCK'` is a variant of `'MEMORY_LOCK'`. A `'TRANSACTION_LOCK'` does nothing when used in conjunction with `'SINGLE_THREADED_HEAP'` or `'MULTI_THREADED_HEAP'` (i.e. it's functionally the same as `'NO_LOCK'`). However, a `'TRANSACTION_LOCK'` behaves like a normal `'MEMORY_LOCK'` when used in conjunction with `'BeginTransaction()'` and `'JoinTransaction()'` on a `'TRANSACTIONAL_HEAP'` or `'DATABASE_HEAP'` (i.e. see chapter 5 above). A `'TRANSACTION_LOCK'` is normally used in situations where there are multiple threads working in parallel on a single transaction.

6.4. The Rockall-DB Library Support Classes

A number of additional miscellaneous classes are also available in the Rockall-DB library. These classes support functionality that may be of use in some situations. All of these Rockall-DB library support classes are show in 'Table-06-10' below:

File Name	Sections	Description
AsynchronousCallbacks.hpp	6.4.1	A class to asynchronously call functions.
AutomaticTransactionScope.hpp	6.4.2	A class to automatically terminate transactions.
AutomaticViewScope.hpp	6.4.3	A class to automatically terminate views.
PlacementNew.hpp	6.4.4	A class to manually call a constructor or destructor.
RockallDelete.hpp	6.4.5	A Rockall replacement for the 'delete' operator.
RockallNew.hpp	6.4.6	A Rockall replacement for the 'new' operator.
RockallTypes.hpp	6.4.7	A collection of standard Rockall types.
VirtualDestructor.hpp	6.4.8	A class to aid the automatic calling of destructors.

Table-06-10

All of the functions listed in 'Table-06-10' are described in detail in the following sections in the order they appear the table above.

6.4.1. The AsynchronousCallbacks Class

The `'ASYNCHRONOUS_CALLBACKS'` class simplifies the management of threads within Rockall-DB. A simple call can be made to asynchronously call a user function which will be made as soon suitable resources are available.

The `'ASYNCHRONOUS_CALLBACKS'` class requires a template parameter to configure it. This template parameter is the type of value to be passed to the asynchronous function. Any supplied value is copied and the copied value passed as the first parameter to the callback function. The callback function may execute for any amount of time and when the callback function ends the associated thread returns to the pool of available threads. An example of the `'ASYNCHRONOUS_CALLBACKS'` class is shown in 'Example-06-28' below:

```
#include <stdio.h>

#include "AsynchronousCallbacks.hpp"
#include "DatabaseHeap.hpp"
#include "RockallTypes.hpp"

void AsynchronousFunction( int *Value )
  { printf( "Asynchronous Function value is %d\n",(*Value) ); }

int main( void )
  {
  DATABASE_HEAP Heap;
  ASYNCHRONOUS_CALLBACKS<int> Callbacks;
  int CPUs = ((int) Callbacks.NumberOfCPUs());
  int Count;

  for ( Count=0;Count < CPUs;Count ++ )
    { Callbacks.Schedule( AsynchronousFunction,Count ); }

  Callbacks.Sleep( 1 );

  Callbacks.WaitAll();
  return 0;
  }
```

Example-06-28

The `'ASYNCHRONOUS_CALLBACKS'` template in this case is `'int'` and so all of the calls to `'AsynchronousFunction()'` for this instance of the class will be passed a pointer to an `'int'` as their first parameter. The `'ASYNCHRONOUS_CALLBACKS'` template can support any structure or type as long as it can be copied with `'memcpy()'`.

The `'NumberOfCPUs()'` function returns the current number of CPUs. The `'Schedule()'` function in this case schedules an asynchronous callback to `'AsynchronousFunction()'` and arranges for a copy of the value of `'Count'` to be passed to it. The call to `'AsynchronousFunction()'` can occur at any point in the future. The precise ordering and the timing of the calls are unpredictable. The `'Sleep()'` function allows the current thread to sleep for a number of milliseconds. Finally, the `'WaitAll()'` function will wait until the callbacks scheduled by `'Schedule()'` have completed for an instance of the `'ASYNCHRONOUS_CALLBACKS'` class.

6.4.2. The AutomaticTransactionScope Class

The `'AUTOMATIC_TRANSACTION_SCOPE'` class is mainly intended for internal use within the Rockall-DB library. Its function is to ensure that any call to `'BeginTransaction'` is matched by a call to `'EndTransaction'` within the scope of a block.

The `'AUTOMATIC_TRANSACTION_SCOPE'` class requires a template parameter to configure it. This template parameter is the type of Rockall-DB heap being used (i.e. in this case a `'DATABASE_HEAP'`).

The `'AUTOMATIC_TRANSACTION_SCOPE'` class partially duplicates the more advanced functionality available in the `'AUTOMATIC_HEAP_SCOPE'` class and is only useful in situations where only this limited functionality is helpful. An example of the `'AUTOMATIC_TRANSACTION_SCOPE'` class is shown in 'Example-06-29' below:

```
#include "AutomaticHeapScope.hpp"
#include "AutomaticTransactionScope.hpp"
#include "DatabaseHeap.hpp"
#include "RockallDelete.hpp"
#include "RockallNew.hpp"
#include "RockallTypes.hpp"

typedef DATABASE_HEAP HEAP_TYPE;

int main( int Count,char *Argument[] )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope( & Heap );
  FILE_ADDRESS Address;
  int Result = 0;
  int *Data;

  if ( (Count == 2) && (Scope.CreateFile( Argument[1] )) )
    {
      // No transaction.
      {
      AUTOMATIC_TRANSACTION_SCOPE<HEAP_TYPE> Transaction( & Scope );

      if ( Transaction.BeginTransaction() )
        {
        // Active transaction.
        if ( ! ROCKALL_NEW<int>::New( & Scope,& Address,& Data ) )
          { Result = 1; }
        }
      }

      // No transaction.
      {
      AUTOMATIC_TRANSACTION_SCOPE<HEAP_TYPE> Transaction( & Heap );

      if ( Transaction.BeginTransaction() )
        {
        // Active transaction.
        if ( ! ROCKALL_DELETE<int>::Delete( & Heap,Address ) )
          { Result = 1; }

        Transaction.EndTransaction();
        // No transaction.
        }
      }

    Scope.CloseFile();
    }

  return Result;
  }
```

<u>Example-06-29</u>

The call to `‘CreateFile’` creates a new `‘DATABASE_HEAP’` file. The constructor to the initial instance of the `‘AUTOMATIC_TRANSACTION_SCOPE’` class is passed the parameter `‘Scope’` which provides access to the `‘Heap’`. Later, `‘Scope’` is also passed as a parameter to the `‘New’` function also to provide access to `‘Heap’`.

The constructor to the later instance of the `‘AUTOMATIC_TRANSACTION_SCOPE’` class is passed the parameter `‘Heap’` directly. Later, `‘Heap’` is also passed as a parameter to the `‘Delete’` function. These two instances demonstrate that `‘Scope’` and `‘Heap’` can often be used interchangeably in various situations.

In the initial block of code a new transaction is created by a call to `‘BeginTransaction’` which terminates when `‘Transaction’` goes out of scope, as there is no call to `‘EndTransaction’`.

In the latter block of code another new transaction is created by a call to `‘BeginTransaction’` but in this case the transaction is terminated early by a call to `‘EndTransaction’`. A transaction may always be terminated early and in this case there will be no further call to `‘EndTransaction’` when `‘Transaction’` goes out of scope.

The main advantage of the `'AUTOMATIC_TRANSACTION_SCOPE'` class relates to its simplicity and performance, which is why it is sometimes used in conjunction with the with `'AUTOMATIC_HEAP_SCOPE'` class in the Rockall-DB library.

6.4.3. The AutomaticViewScope Class

The `'AUTOMATIC_VIEW_SCOPE'` class is mainly intended for internal use within the Rockall-DB library. Its function is to ensure that any call to `'ExclusiveView'` or `'View'` is matched by a call to `'EndExclusiveView'` or `'EndView'` respectively within the scope of a block.

The `'AUTOMATIC_VIEW_SCOPE'` class requires a template parameter to configure it. This template parameter is the type of Rockall-DB heap being used (i.e. in this case a `'DATABASE_HEAP'`).

An example of the `'AUTOMATIC_VIEW_SCOPE'` class is shown in 'Example-06-30' below:

```
#include "AutomaticHeapScope.hpp"
#include "AutomaticTransactionScope.hpp"
#include "AutomaticViewScope.hpp"
#include "DatabaseHeap.hpp"
#include "RockallDelete.hpp"
#include "RockallNew.hpp"
#include "RockallTypes.hpp"

typedef DATABASE_HEAP HEAP_TYPE;

int main( int Count,char *Argument[] )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope( & Heap );
  FILE_ADDRESS Address;
  int Result = 0;
  int *Data;

  if ( (Count == 2) && (Scope.CreateFile( Argument[1] )) )
    {
      //  No transaction and no view.
      {
      AUTOMATIC_TRANSACTION_SCOPE<HEAP_TYPE> Transaction( & Scope );

      if ( Transaction.BeginTransaction() )
        {
        // Active transaction but no view.
        if ( ! ROCKALL_NEW<int>::New( & Scope,& Address,& Data ) )
          { Result = 1; }
        }
      }

      //  No transaction and no view.
      {
      AUTOMATIC_TRANSACTION_SCOPE<HEAP_TYPE> Transaction( & Scope );

      if ( Transaction.BeginTransaction() )
        {
          // Active transaction but no view.
          {
```

```
          AUTOMATIC_VIEW_SCOPE<HEAP_TYPE> View( & Scope );

          if ( View.ExclusiveView( Address,((void**) & Data) ) )
            {
            //  Active transaction and exclusive view.
            }
          }

          // Active transaction but no view.
          {
          AUTOMATIC_VIEW_SCOPE<HEAP_TYPE> View( & Scope );

          if ( View.View( Address,((void**) & Data) ) )
            {
            //  Active transaction and shared view.
            }
          }

          // Active transaction but no view.
          {
          AUTOMATIC_VIEW_SCOPE<HEAP_TYPE> View( & Heap );

          if ( View.ExclusiveView( Address,((void**) & Data) ) )
            {
            //  Active transaction and exclusive view.
            View.Release();
            // Active transaction but no view.
            }
          }
        }

        //  No transaction and no view.
        {
        AUTOMATIC_TRANSACTION_SCOPE<HEAP_TYPE> Transaction( & Heap );

        if ( Transaction.BeginTransaction() )
          {
          // Active transaction but no view.
          if ( ! ROCKALL_DELETE<int>::Delete( & Heap,Address ) )
            { Result = 1; }

          Transaction.EndTransaction();
          //  No transaction and no view.
          }
        }
      }

    Scope.CloseFile();
    }

  return Result;
  }
```

Example-06-30

The call to `'CreateFile'` creates a new `'DATABASE_HEAP'` file. The constructor to the initial instance of the `'AUTOMATIC_TRANSACTION_SCOPE'` class is passed the parameter `'Scope'` which provides access to the `'Heap'`. Later, `'Scope'` is also passed as a parameter to the `'New'` function to also provide access to `'Heap'`.

The constructor to the last instance of the `'AUTOMATIC_TRANSACTION_SCOPE'` class is passed the parameter `'Heap'` directly. Later, `'Heap'` is also passed as a parameter to the `'Delete'` function. These two instances demonstrate that `'Scope'` and `'Heap'` can often be used interchangeably in various situations.

In the central block of code a new transaction is created by a call to `'BeginTransaction'` which terminates when `'Transaction'` goes out of scope, as there is no call to `'EndTransaction'`.

In the central block of code also contains three blocks of code that create views. The constructors for the first two instances of the `'AUTOMATIC_VIEW_SCOPE'` class are passed the parameter `'Scope'` which allows them to access `'Heap'`. The final instance of the `'AUTOMATIC_VIEW_SCOPE'` class is passed the parameter `'Heap'` directly. These three instances again demonstrate that `'Scope'` and `'Heap'` can often be used interchangeably in various situations.

The initial instance of the `'AUTOMATIC_VIEW_SCOPE'` class creates an `'ExclusiveView'` with which terminates when `'View'` goes out of scope, as there is no call to `'Release'`.

The next instance of the `'AUTOMATIC_VIEW_SCOPE'` class creates a shared `'View'` with which terminates when `'View'` goes out of scope, as there is no call to `'Release'`.

The final instance of the `'AUTOMATIC_VIEW_SCOPE'` class creates an `'ExclusiveView'` but in this case the view is terminated early by a call to `'Release'`. A view may always be terminated early and in this case there will be no further call to `'EndExclusiveView'` when `'View'` goes out of scope.

6.4.4. The LinkedList Class

The `'LINKED_LIST'` class supports in-memory linked lists and works with all Rockall-DB heaps except for `'DATABASE_HEAP'`. All of the elements are doubly linked so new elements can easily be added or removed anywhere in a list. An example of the `'LINKED_LIST'` class is shown in 'Example-06-31' below:

```
#include <stdio.h>

#include "LinkedList.hpp"
#include "RockallTypes.hpp"

//
//  A linked list element.
//
class ELEMENT : public LINKED_LIST<ELEMENT>
  {
  public:
    int Value;

    ELEMENT( int NewValue = 0 )
      { Value = NewValue; }

    ~ELEMENT( void )
      { Value = 0; }
  };

//
//  A linked list header.
//
typedef LINKED_LIST<ELEMENT> HEADER;

int main( void )
  {
  ELEMENT Element[4] = { 0,1,2,3 };
  ELEMENT *Current;
  HEADER Header;

  //
  //  Insert 'Element[0]' at the head of the list
  //  and 'Element[3]' at the end of the list.
  //
  Element[0].InsertFirst( & Header );
  Element[3].InsertLast( & Header );

  //
  //  Insert 'Element[1]' after 'Element[0]' then
  //  insert 'Element[2]' before ' Element[3]' in
  //  the list.
  //
  Element[1].InsertAfter( & Header,& Element[0] );
  Element[2].InsertBefore( & Header,& Element[3] );

  //
  //  We iterate the list in 'Forwards()' order.
  //
  for
      (
      Current = Header.First();
      ! Current -> End();
      Current = Current -> Forwards()
      )
    {
    //
    //  We visit each list 'ELEMENT' in 'Forwards()'
    //  order.
    //

    printf( "%d\n",Current -> Value );
    }

  //
  //  We iterate the list in 'Backwards()' order.
  //
  for
      (
      Current = Header.Last();
      ! Current -> End();
      Current = Current -> Backwards()
      )
    {
    //
    //  We visit each list 'ELEMENT' in 'Backwards()'
    //  order.
    //
    printf( "%d\n",Current -> Value );
    }

  //
  //   We unlink all the elements from the end of
  //   the list until it is empty.
  //
  for
      (
      Current = Header.Last();
      ! Current -> End();
      Current = Header.Last()
      )
    { Current -> Unlink( & Header ); }

  return 0;
  }
```

Example-06-31

A call to `'InsertFirst'` inserts a `'LINKED_LIST'` element at the start of a list whereas a call to `'InsertLast'` inserts an element at the end of a list. These calls must be supplied with a pointer to the list 'Header' as either the head or tail of the list is guaranteed to change.

A call to `'InsertAfter'` inserts a new `'LINKED_LIST'` element after an existing element whereas a call to `'InsertBefore'` inserts a `new` element before an existing element. These calls must be supplied with a pointer to the list 'Header' just in case the head or tail of the list needs to change along with a pointer to an existing element in the list.

The first loop walks along the list in from head to tail printing each value as it goes. The output from each execution of this loop will be "0", "1", "2" and "3" respectively.

The next loop walks along the list in from tail to head printing each value as it goes. The output from each execution of this loop will be "3", "2", "1" and "0" respectively.

Finally, a call to `'Unlink'` removes an existing element from the list. This call must be supplied a pointer to the list 'Header' just in case the head or tail of the list needs to change. The final loop unlinks the last element on each cycle until the list is empty.

6.4.5. The PlacementNew Header

The `'PlacementNew.hpp'` header contains a number of macros that allow class constructors and destructors to be called manually. Typically, the `'ROCKALL_NEW'` and `'ROCKALL_DELETE'` classes would be used instead of this functionality, as they automatically call the related constructors and destructors (i.e. see sections 6.4.3 and 6.4.4 below). Nonetheless, the `'PlacementNew.hpp'` header is available for general use if needed. An example of the `'PlacementNew.hpp'` header is shown in 'Example-06-32' below:

```
#include "AutomaticHeapScope.hpp"
#include "AutomaticTransactionScope.hpp"
```

```
#include "DatabaseHeap.hpp"
#include "PlacementNew.hpp"
#include "RockallTypes.hpp"

typedef DATABASE_HEAP HEAP_TYPE;
typedef struct { int Value; } DATA;

int main( int Count,char *Argument[] )
  {
  HEAP_TYPE Heap;
  AUTOMATIC_HEAP_SCOPE<HEAP_TYPE> Scope( & Heap );
  FILE_ADDRESS Address;
  DATA *Data;

  if ( (Count == 2) && (Scope.CreateFile( Argument[1] )) )
    {
      // No transaction.
      {
      AUTOMATIC_TRANSACTION_SCOPE<HEAP_TYPE> Transaction( & Scope );

      if ( Transaction.BeginTransaction() )
        {
        // Active transaction.
        if ( Scope.New( & Address,((void**) & Data),sizeof(DATA) ) )
          {
          PLACEMENT_NEW( Data,DATA );
          // Active allocation ready for use.
          PLACEMENT_DELETE( Data,DATA  );

          Scope.Delete( Address );
          }
        }
      }

    // No transaction.
    Scope.CloseFile();
    }

  return 0;
  }
```

<u>Example-06-32</u>

The call to `'CreateFile'` creates a new `'DATABASE_HEAP'` file. The constructor to the initial instance of the `'AUTOMATIC_TRANSACTION_SCOPE'` class is passed the parameter `'Scope'` which is used to access `'Heap'`.

In the main block of code a new transaction is created by a call to `'BeginTransaction'` which terminates when `'Transaction'` goes out of scope, as there is no call to `'EndTransaction'`.

A call is made to the native Rockall-DB `'New'` (i.e. which does not call the related constructor) to make a new memory allocation followed by a call to `'PLACEMENT_NEW'` to call the related constructor.

Later, a call to `'PLACEMENT_DELETE'` is made to call the related destructor. Next, a call is made to the native Rockall-DB `'Delete'` (i.e. which does not call the related destructor) to free the memory allocation.

6.4.6. The RockallDelete Class

The `'ROCKALL_DELETE'` class is the Rockall-DB alterative to the C++ `'delete'` operator.

The `'ROCKALL_DELETE'` class requires a template parameter to configure it. This template parameter is the type of memory allocation being deleted (i.e. in this case an `'int'`).

An example of the `'ROCKALL_DELETE'` class is shown in 'Example-06-33' below:

```
#include "SingleThreadedHeap.hpp"
#include "RockallDelete.hpp"
#include "RockallNew.hpp"
#include "RockallTypes.hpp"

int main( void )
  {
  SINGLE_THREADED_HEAP Heap;
  FILE_ADDRESS Address;
  int *Data;
  FILE_SIZE Space;

  if
      (
      ROCKALL_NEW<int>::New
          (
          & Heap,
          & Address,
          & Data,
          1,
          & Space,
          False,
          True
          )
      )
    {
    ROCKALL_DELETE<int>::Delete
      (
      & Heap,
      Address,
      1,
      True
      );
    }

  return 0;
  }
```

Example-06-33

The `'ROCKALL_DELETE'` class in 'Example-06-33' supports a number of variants of the function `'Delete'`.

The initial parameter is pointer to a Rockall-DB `'Heap'` or instance of `'AUTOMATIC_HEAP_SCOPE'`.

The next parameter is either a `'FILE_ADDRESS'` or data pointer previously returned from `'New'` in the `'ROCKALL_NEW'` class.

The next parameter is optional and is a count of instances of the type. The default value is `'1'`. This parameter must match the corresponding parameter on the call to `'New'` in the `'ROCKALL_NEW'` class when the allocation was made. This parameter controls the number of times the destructor is called before the allocation is deleted. If the value is incorrect the destructor will be called too many or too few times and this may damage other nearby data structures in the process.

The final parameter is optional and will zero the memory allocation just before it is deleted if set to `'True'` or do nothing if set to `'False'` (i.e. see `'Delete'` in section 5.2.1). The default value is `'False'`.

The `'ROCKALL_DELETE'` class is designed to closely resemble the format and functionality of the Rockall-DB `'Delete'` function (see section 5.2.1 for further details).

6.4.7. The RockallNew Class

The `'ROCKALL_NEW'` class is the Rockall-DB alterative to the C++ `'new'` operator.

The `'ROCKALL_NEW'` class requires a template parameter to configure it. This template parameter is the type of memory allocation being allocated (i.e. in this case an `'int'`).

An example of the `'ROCKALL_NEW'` class is shown in 'Example-06-34' below:

```
#include "SingleThreadedHeap.hpp"
#include "RockallDelete.hpp"
#include "RockallNew.hpp"
#include "RockallTypes.hpp"

int main( void )
  {
  SINGLE_THREADED_HEAP Heap;
  FILE_ADDRESS Address;
  int *Data;
  FILE_SIZE Space;

  if
      (
      ROCKALL_NEW<int>::New
          (
          & Heap,
          & Address,
          & Data,
          1,
          & Space,
          False,
          True
          )
      )
    {
    ROCKALL_DELETE<int>::Delete
      (
      & Heap,
      Address,
      1,
      True
      );
    }

  return 0;
  }
```

Example-06-34

The `'ROCKALL_NEW'` class in 'Example-06-34' supports a number of variants of the function `'New'`.

The initial parameter is a pointer to a Rockall-DB `'Heap'` or instance of `'AUTOMATIC_HEAP_SCOPE'`.

The next parameter is optional and if present must point to a `'FILE_ADDRESS'`. If the associated `'Heap'` is a `'DATABASE_HEAP'` this location will contain the `'FILE_ADDRESS'` of the allocation (i.e. see `'New'` in section 5.2.6).

The next parameter is a pointer to the location where the address of the new memory allocation can be stored (i.e. a pointer to an `'int*'` in this case).

The next parameter is optional and is used to help allocate arrays. Let's say the value `'3'` was supplied and the template parameter was type `'int'`. In this case, enough space would be allocated for an array of type `'int[3]'` and the constructor for `'int'` would be called for each element in the array. The default value is `'1'`.

The next parameter is optional and will return the actual amount of space allocated if it is supplied with a pointer to a `'FILE_SIZE'` (i.e. see `'New'` in section 5.2.6).

The next parameter is optional and will set the `'Destroy'` flag (i.e. see `'New'` in section 5.2.6).

The final parameter is optional and will zero the memory allocation if set to `'True'` or do nothing if set to `'False'` (i.e. see `'New'` in section 5.2.6). The default value is `'False'`.

The `'ROCKALL_NEW'` class is designed to closely resemble the format and functionality of the Rockall-DB `'New'` function (see section 5.2.6 for further details).

6.4.8. The RockallTypes Header

The `'RockallTypes.hpp'` header contains a number of common Rockall-DB constants and types. The structure of the `'RockallTypes.hpp'` header is shown in 'Example-06-35' below:

```
typedef ROCKALL::BOOLEAN                                    BOOLEAN;
typedef ROCKALL::CHAR                                       CHAR;
typedef ROCKALL::FILE_ADDRESS                               FILE_ADDRESS;
typedef ROCKALL::FILE_SIZE                                  FILE_SIZE;
typedef ROCKALL::SBIT16                                     SBIT16;
typedef ROCKALL::SBIT32                                     SBIT32;
typedef ROCKALL::SBIT64                                     SBIT64;
typedef ROCKALL::WCHAR                                      WCHAR;
```

```
static const BOOLEAN False                              = (ROCKALL::False);
static const BOOLEAN True                               = (ROCKALL::True);

static const SBIT32 NoRegion                            = (ROCKALL::NoRegion);

static const SBIT32 RockallDatabaseHeap                 = (ROCKALL::RockallDatabaseHeap);
static const SBIT32 RockallMultiThreadedHeap            = (ROCKALL::RockallMultiThreadedHeap);
static const SBIT32 RockallSingleThreadedHeap           = (ROCKALL::RockallSingleThreadedHeap);
static const SBIT32 RockallTransactionalHeap            = (ROCKALL::RockallTransactionalHeap);

static const SBIT32 RockallFileSupport                  = (ROCKALL::RockallFileSupport);
static const SBIT32 RockallLockSupport                  = (ROCKALL::RockallLockSupport);
static const SBIT32 RockallMemorySupport                = (ROCKALL::RockallMemorySupport);
static const SBIT32 RockallTransactionalSupport         = (ROCKALL::RockallTransactionalSupport);
```

Example-06-35

6.4.9. The VirtualDestructor Class

The `'VIRTUAL_DESTRUCTOR'` class does not contain any functional code. All it does is to make the destructor of any class that inherits from it `'virtual'`. It also ensures that the associated `'v-pointer'` is the first entry in the `'v-table'` so that Rockall-DB can find it if necessary. An example of the `'VIRTUAL_DESTRUCTOR'` class and its associated functionality is available in chapter 3 in ‘Example-03-09’ and ‘Example-03-10’.

7. The Rockall-DB Build

A Rockall-DB build consists of a single directory called `'Build'` which contains a number of sub-directories. These sub-directories are dedicated to specific areas of the product and are outlined below.

7.1. The Documents Directory

The `'Documents'` directory contains all of the documentation related to Rockall-DB, which will typically include a copy of this manual.

7.2. The Examples Directory

The `'Examples'` directory contains a collection of Rockall-DB examples grouped together into a number of sub-directories. These sub-directories are dedicated to specific areas of the product and are outlined below.

7.2.1. The Boggle Directory

The `'Boggle'` directory contains solver for the well-known game 'Boggle'. In this game 16 dice each with letters on them are shaken and arranged into a 4 by 4 grid. The goal is to find as many words in the 4 by 4 grid by using adjacent letters but without reusing any of the letters. All of the source code is supplied in the `'Code'` directory and an example dictionary is supplied in the `'Data'` directory.

It is left to the reader to compile the C++ code in the `'Code'` directory into a suitable executable. A project is available in the directory 'VisualStudio' to build this and a number of other examples if the reader has access to compatible tools. The resulting program can then be run to build an example dictionary database as shown in 'Example-07-01' below:

```
Boggle.exe –c Dictionary.db Dictionary.txt
```

Example-07-01

In 'Example-07-01', `'Boggle.exe'` is the name of the new executable program, `'Dictionary.db'` is an arbitrary name for the new database and `'Dictionary.txt'` is the name of the example dictionary supplied in the `'Data'` directory.

When the above steps have been completed the new database (i.e. `'Dictionary.db'`) can be used to play the game as shown in 'Example-07-02' below:

```
Boggle.exe –p Dictionary.db
```

Example-07-02

Following the steps in 'Example-07-02' will typically produce output similar to 'Example-07-03' below:

```
Please type in the 'Boggle' board.  All you need to type in is the
letters on each line of the board without any spaces.  The letters
can be in either capitals or lower case.  A new line should be typed
at the end of each row of the board.  All the rows should contain the
same number of letters.  When you have typed in all the rows on the
board simply enter a blank line to denote the end of the board.

Please enter a 'Boggle' board now:
```

```
wert
sdfg
zxcv
qwer

The 'Boggle' board size is 4 x 4:

The 'Boggle' words are:

Score 2 for 'crew'
Score 1 for 'dew'
Score 2 for 'drew'
Score 1 for 'fed'
Score 1 for 'few'
Score 2 for 'grew'
Score 1 for 'red'
Score 1 for 'sew'
Score 1 for 'vex'
Score 2 for 'wert'

The total score is 14.
```

Example-07-03

7.2.1. The Book Directory

The `'Book'` directory contains an example of a program that builds a new database from all the words in a book. The new database can then be used to search for arbitrary keywords in the book to find any related paragraphs. All of the source code is supplied in the `'Code'` directory and an example book is supplied in the `'Data'` directory.

It is left to the reader to compile the C++ code in the `'Code'` directory into a suitable executable. A project is available in the directory 'VisualStudio' to build this and a number of other examples if the reader has access to compatible tools. The resulting program can then be run to build an example database as shown in 'Example-07-03' below:

```
Book.exe -c Database.db Book.txt
```

Example-07-03

In 'Example-07-03', `'Book.exe'` is the name of the new executable program, `'Database.db'` is an arbitrary name for the new database and `'Book.txt'` is the name of the example book supplied in the `'Data'` directory.

Following the steps in 'Example-07-03' will typically produce output similar to 'Example-07-05' below:

```
Summary: 789638 words (12557 unique)
```

Example-07-05

When the above steps have been completed the new database (i.e. `'Database.db'`) can then be searched as shown in 'Example-07-06' below:

```
Book.exe -f Database.db door knock
```

Example-07-06

In 'Example-07-06', `'Book.exe'` is the name of the executable program, `'Database.db'` is the new database built in 'Example-07-04' and `'door'` and `'knock'` are arbitrary keywords to find in the book.

The example book supplied in the `'Data'` directory is a copy of the 'King James' version of the Bible which dates back to 1611. This book was selected because God is considerably more generous with his copyright terms than many other authors. Regardless, any book with a similar format could be substituted in its place.

Following the steps in 'Example-07-06' will typically produce output similar to 'Example-07-07' below:

```
Matches: 2 matching reference(s) found.


Title: 'The Gospel According to Saint Luke'

Reference:
'13:25 When once the master of the house is risen up, and hath shut
to the door, and ye begin to stand without, and to knock at the door,
saying, Lord, Lord, open unto us; and he shall answer and say unto
you, I know you not whence ye are: 13:26 Then shall ye begin to say,
We have eaten and drunk in thy presence, and thou hast taught in our
streets.'


Title: 'The Revelation of Saint John the Devine'

Reference:
'3:20 Behold, I stand at the door, and knock: if any man hear my
voice, and open the door, I will come in to him, and will sup with
him, and he with me.'
```

Example-07-07

It is left to the reader to try other keywords, examine the C++ source code and from this starting point draw benefit from the example.

7.2.2. The Dump Directory

The `'Dump'` directory contains an example of a program that converts the contents of an arbitrary Rockall-DB `'DATABASE_HEAP'` file in printable dump format. It achieves this by walking the related `'DATABASE_HEAP'` and outputting a string for every active allocation it encounters.

It is left to the reader to compile the C++ code in the `'Code'` directory into a suitable executable. A project is available in the directory 'VisualStudio' to build this and a number of other examples if the reader has access to compatible tools. The resulting program can then be run on a suitable file as shown in 'Example-07-05' below:

```
Dump.exe Database.db
```

Example-07-08

In ‘Example-07-08’, `‘Dump.exe’` is the name of the new executable program and `‘Database.db’` is an arbitrary file name relating to a suitable file (i.e. `‘Database.db’` from ‘Example-07-04’).

Following the steps in ‘Example-07-08’ will typically produce output similar to ‘Example-07-09’ below:

```
0x003af0a0 (0x01bdf0a0) 32 : 0000000000000001000000000000000700676e696b6c61740000000000000000 : ????????????????talking?????????
0x003af0c0 (0x01bdf0c0) 32 : 0000000000000001000000000000000500000072616d61740000000000000000 : ????????????????tamar???????????
0x003af0e0 (0x01bdf0e0) 32 : 0000000000000001000000000000000700646569727261740000000000000000 : ????????????????tarried?????????
0x003af100 (0x01bdf100) 32 : 0000000000000001000000000000000500000079727261740000000000000000 : ????????????????tarry???????????
0x003af120 (0x01bdf120) 32 : 0000000000000001000000000000000500000068616265740000000000000000 : ????????????????tebah???????????
0x003af140 (0x01bdf140) 32 : 00000000000000010000000000000004000000006c6c65740000000000000000 : ????????????????tell????????????
0x003af160 (0x01bdf160) 32 : 0000000000000001000000000000000400000000616d65740000000000000000 : ????????????????tema????????????
0x003af180 (0x01bdf180) 32 : 00000000000000010000000000000005000000006e616d6574000000000000000 : ????????????????teman???????????
0x003af1a0 (0x01bdf1a0) 32 : 00000000000000010000000000000006000000696e616d65740000000000000000 : ????????????????temani??????????
0x003af1c0 (0x01bdf1c0) 32 : 0000000000000001000000000000000500000074706d65740000000000000000 : ????????????????tempt???????????
0x003af1e0 (0x01bdf1e0) 32 : 000000000000000100000000000000030000000000006e65740000000000000000 : ????????????????ten?????????????
0x003af200 (0x01bdf200) 32 : 0000000000000001000000000000000600007265646e65740000000000000000 : ????????????????tender??????????
0x003af220 (0x01bdf220) 32 : 0000000000000001000000000000000400000000746e65740000000000000000 : ????????????????tent????????????
0x003af240 (0x01bdf240) 32 : 0000000000000001000000000000000500000068746e65740000000000000000 : ????????????????tenth???????????
0x003af260 (0x01bdf260) 32 : 0000000000000001000000000000000500000073746e65740000000000000000 : ????????????????tents???????????
0x003af280 (0x01bdf280) 32 : 0000000000000001000000000000000500000068617265740000000000000000 : ????????????????terah???????????
0x003af2a0 (0x01bdf2a0) 32 : 000000000000000100000000000000060000726f727265740000000000000000 : ????????????????terror??????????
0x003af2c0 (0x01bdf2c0) 32 : 0000000000000001000000000000000700687361686168740000000000000000 : ????????????????thahash?????????
0x003af2e0 (0x01bdf2e0) 32 : 00000000000000010000000000000004000000006e6168740000000000000000 : ????????????????than????????????
0x003af300 (0x01bdf300) 32 : 0000000000000001000000000000000400000000746168740000000000000000 : ????????????????that????????????
0x003af320 (0x01bdf320) 32 : 0000000000000001000000000000000300000000006568740000000000000000 : ????????????????the?????????????
0x003af340 (0x01bdf340) 32 : 0000000000000001000000000000000400000000656568740000000000000000 : ????????????????thee????????????
0x003af360 (0x01bdf360) 32 : 0000000000000001000000000000000500000072696568740000000000000000 : ????????????????their???????????
0x003af380 (0x01bdf380) 32 : 00000000000000010000000000000004000000006d6568740000000000000000 : ????????????????them????????????
0x003af3a0 (0x01bdf3a0) 32 : 00000000000000010000000000000004000000006e6568740000000000000000 : ????????????????then????????????
0x003af3c0 (0x01bdf3c0) 32 : 000000000000000100000000000000060000656336e6568740000000000000000 : ????????????????thence??????????
0x003af3e0 (0x01bdf3e0) 32 : 0000000000000001000000000000000500000065726568740000000000000000 : ????????????????there???????????
0x003af400 (0x01bdf400) 32 : 0000000000000001000000000000000700796265726568740000000000000000 : ????????????????thereby?????????
0x003af420 (0x01bdf420) 32 : 00000000000000010000000000000007006e6965726568740000000000000000 : ????????????????therein?????????
0x003af440 (0x01bdf440) 32 : 0000000000000001000000000000000700666f65726568740000000000000000 : ????????????????thereof?????????
0x003af460 (0x01bdf460) 32 : 00000000000000010000000000000007006e6f65726568740000000000000000 : ????????????????thereon?????????

(*** A large section of the dump has been removed ***)

Totals: Active 17202 (4592776 bytes), Idle 1853 (846184 bytes)
```

Example-07-09

Clearly, a dump would typically be much larger than ‘Example-07-09’ but to save space only part of the dump appears in this manual.

It is left to the reader to examine the C++ source code and from this starting point draw benefit from the example. Nonetheless, this example is actually a practical tool that may prove useful in some situations when using Rockall-DB.

7.2.3. The Manual Directory

The `‘Manual’` directory contains most of the programming examples provided in this manual and other documents relating to Rockall-DB. Most of the examples are free-standing C++ programs and can be easily compiled into executable code. These examples are intended as demonstrations and as starting points for developing more complex programs. A project is available in the directory ‘VisualStudio’ to build this and a number of other examples if the reader has access to compatible tools.

7.2.4. The MultiLanguage Directory

The whole of Rockall-DB was originally written in Microsoft C++ and targeted at the Microsoft Windows operating system. These choices were made because at the time very few programming languages were flexible enough, fast enough or powerful enough to support Rockall-DB (i.e. features such as templates) and because Microsoft Windows was a widely accepted platform with advanced support for features such as asynchronous I/O, multi-threading and wide character support. Nonetheless, from the beginning Rockall-DB was intended to support multiple programming languages and to be portable to multiple operating systems.

The `‘MultiLanguage’` directory contains the first steps along this pathway towards multi-language support and multi-operating system support. The current files in this directory are `'MultiLanguage.hpp'` and `'MultiLanguage.cpp'`. These files convert a number of the Rockall-DB interfaces into standard 'C' type functions suitable for calls from other programming languages such as C#, COBOL, Fortran, Java and Pascal. Any user that is keen to tailor this interface to any of the above programming languages (or other programming languages) should contact Rockall Software Ltd for additional support and to optionally get these tailored interfaces rolled into later versions of Rockall-DB.

A pre-built version of `'MultiLanguage.hpp'` and `'MultiLanguage.cpp'` has been compiled into a Dynamic Link Library (DLL) and appears along with all the other Rockall-DB object files. The names of the related files are `'MultiLanguage.lib'` and `'MultiLanguage.dll'` and can be used directly without modification by users. The source code files `'MultiLanguage.hpp'` and `'MultiLanguage.cpp'` are provided so that these interfaces can be adjusted, extended or modified to fit the precise requirements of users.

7.3. The Unix Directory

The `'Unix'` directory will contain builds of Rockall-DB for versions of UNIX when they become available.

7.4. The Windows Directory

The `'Windows'` directory contains a build of Rockall-DB for Microsoft Windows and contains a number of sub-directories. These sub-directories are dedicated to specific areas of the product and are outlined below.

7.4.1. The Include Directory

The `'Include'` directory contains all the header files relating to the Rockall-DB core and Rockall-DB library. Any of these header files can be included into suitable source code, as demonstrated in examples in chapters 3 and 4. An explanation of all of the header files is available in chapter 6 above.

There are a number of sub-directories within the `'Include'` directory and typically all of these sub-directories should be added into the C++ `'Include'` path to fully utilize the product. Alternatively, the contents of these sub-directories could be copied into an alternative directory that is already on the `'Include'` path and be used in this way.

A significant portion of Rockall-DB has been coded in a C++ `'namespace'` called `'ROCKALL'` to mitigate symbol clashes. Consequently, it may occasionally be necessary to refer to functions or types in Rockall-DB by their fully qualified names (e.g. such as `'ROCKALL::BOOLEAN'`). If this becomes tiresome it can easily be avoided by a declaration such as shown in 'Example-07-10' below:

```
typedef ROCKALL::BOOLEAN MY_NAME;
```

Example-07-10

An additional preprocessor flag called `'DISABLE_ROCKALL_GLOBAL_TYPES'` is also available to remove some of the global Rockall-DB types and so as mitigate symbol clashes.

7.4.2. The Library Directory

The `‘Library’` directory contains all the various native code builds of Rockall-DB. The `‘Library’` directory contains the `‘Win32’` and the `‘x64’` sub-directories, which contain the various 32-bit and 64-bit builds of the product respectively. The `‘Win32’` and `‘x64’` directories contain the `‘Debug’` and `‘Release’` sub-directories, which contain a debugging build and a fully optimized build of Rockall-DB respectively. Additionally, there may sometimes be a `‘Trace’` sub-directory which contains an extending `‘Debug’` build of Rockall-DB that includes additional code for tracing memory leaks.

All of the builds are provided as a linkable library file (i.e. `‘*.lib’`) along with an associated dynamic link library file (i.e. `‘*.dll’`) and optionally a symbol file (i.e. `‘*.pdb’`). The linkable library from the selected build should be included in the compilation process in usual way. The related dynamic link library along with any associated symbol file should be placed on dynamic load path, optionally with any other necessary executables (i.e. such as any other necessary `‘*.dll’`, `‘*.exe’` or `‘*.pdb’` files).

There is no need to register or install anything with Rockall-DB. All that is required is to link with the appropriate linkable library file (i.e. include the appropriate `‘*.lib’` when linking the final executable) and to have the dynamic link library file available in the load path for executables. Unfortunately, the precise mechanism for this varies for specific compilers and languages but a typical example for C or C++ might be as shown in ‘Example-07-11’ below:

```
cc MyProgram.cpp Rockall-DB.lib
```

Example-07-11

7.5. A Quick Start Guide

The easiest way to get Rockall-DB up and running is to simply copy all the necessary headers and libraries into the same directory and then compile the application and run it. Let’s use the ‘Dump’ program discussed above as an example. We first need to copy the source code of the program (i.e. `‘Dump.cpp’`) into a newly created empty directory. We then need to copy all the headers in all `‘include’` subdirectories into the same directory. Finally, we need to copy the appropriate version of `‘RockallDB.lib’` and `‘RockallDB.dll’` (i.e. the 32 bit or 64 bit version) into the same directory.

Unfortunately, the precise mechanism for doing the compilation varies for specific compilers and languages but a typical example for C or C++ might be as shown in ‘Example-07-12’ below:

```
cc Dump.cpp Rockall-DB.lib
```

Example-07-12

The above command should compile `‘Dump.cpp’` and link it with `‘RockallDB.dll’` to create an executable version of it. This should work because everything needed to complete the compilation is in the same directory (i.e. all the include files, libraries and dynamic link libraries). Finally, the new executable can be typically run by typing a command like the one shown in ‘Example-07-13’ below:

```
Dump.exe
```

Example-07-13

8. Advice and Guidance

The ‘Advice and Guidance’ section is intended to help users get the most out of Rockall-DB. A wide variety of areas are discussed to highlight various aspects of the product.

8.1. Architecture

A number of architecture related topics are covered in this section. These topics are presented in alphabetical order and are not necessarily closely related.

8.1.1. The Design and Implementation of Multiuser Systems

A number of common architectural and design patterns have been developed over the last few decades. A few well-known examples are ‘Client/Server’, ‘Service Oriented Architectures’ and ‘3-tier Architectures’. Some might imply that Rockall-DB is not particularly good at supporting some of these historical architectures and that it might be helpful if Rockall-DB provided a remote access layer like many traditional databases (i.e. an interface like ‘ODBC’). While something of this nature may be provided in later versions of Rockall-DB it seems worthwhile to briefly discuss reasons why was not included initially and the downsides of such technologies.

A remote access technology (i.e. like ‘ODBC’) allows a program to send ‘SQL’ statements to a remote database and to receive back the associated result sets. If we consider the resources required to send a ‘SQL’ statement to a remote machine, receive it on the remote machine, execute the necessary context switches, execute the ‘SQL’ statement, marshal the result set, send the results back across the network and then return them to the calling process, then it is obvious that all such technologies can never be very efficient, responsive or scalable. Moreover, with Rockall-DB it is often possible to compute and process an entire result set before remote access technologies can even send the first network packet. Consequently, a conscious decision was made to deprecate such models in Rockall-DB as they are the sworn enemies of performance. Nonetheless, this poses the question as to what suitable alternative models are available to replace them.

A common suggestion is to move all of the components of a system onto a single server and ideally execute the entire system within a single process. Clearly, for reliability reasons such a node would need to be part of a cluster and some additional changes may be needed for security. Nonetheless, this is entirely practical and reasonable in most cases. Clearly, there will be howls of protest at this point and so we will now try to deal with most of these objections.

Let’s begin with a challenge. The challenge is to take an existing multi-node system and to move it onto a single node and measure the change in performance. There is no need to modify the system in any way or remove any networking code. Simply load the entire system on to a suitable single node and measure the change in performance. It is not uncommon to see a performance improvement of between 2 and 5 fold even when using significantly less hardware. This often catches developers unawares because few people fully understand the costs and overheads associated with networking. A common objection is that such a system is not scalable. This may have been a concern a few years ago but today a single node to capable of supporting thousands of users and consisting of 24+ cores, gigabytes of DRAM and terabytes of storage costs just a few thousand dollars. Consequently, building an appropriate 2-node cluster is now cheap, easy and cost effective. Furthermore, day-to-day management is far easier, as is disaster recovery. Moreover, replacing a traditional 'SQL' oriented database with Rockall-DB only amplifies this effect driving the levels of performance and scalability even higher. Another common objection is that the database is no longer a single entity on a single machine and so this creates a different type of complexity. However, this is already the case in many larger installations and it is common to have multiple databases managed by separate machines but stored in a common storage system (i.e. a 'SAN'). Furthermore, separating databases in this way typical reduces data fragmentation and so again enhances performance and scalability. A final concern is that merging databases into front-end web servers (or the like) introduces significant security risks. Again, such problems can be overcome using various methods (see section 8.4 for a suggested solution) and so this is not a major concern. Moreover, if properly designed the final result can significantly improve security rather than put it at risk.

In summary, Rockall-DB often challenges existing architectures, design patterns and thinking and calls into questions their intellectual basis. Moreover, Rockall-DB often shamelessly strives for performance, responsiveness, scalability and simplicity even when this might be unpopular. Consequently, the philosophy of Rockall-DB deprecates patterns of thinking that lead to poor architecture and design patterns to actively encourage better architectures and designs. It is believed that this stand will ultimately lead to better products.

8.1.2. The Design of Storage Structures

A nice feature in many existing transactional databases is the ability to add new columns into existing rows within a database table. While this can be quite helpful the performance implications of this feature can be somewhat dire.

A 'SELECT' statement in 'SQL' allows a user to choose which of the available fields they would like to appear in the final result set. Regardless, all of the available fields are still typically read from storage and usually copied into the required position in the final result set.

All of the above implies a very significant performance overhead and so is not supported in Rockall-DB. Nonetheless, many of the associated benefits can still be obtained in Rockall-DB by making a small design change. All that is needed is a unique `'Type'` field as the first element in any relevant class or data structure. While this might seem like a fairly modest requirement there are significant benefits associated with it. A selection of these benefits is outlined below:

1. When using `'Update'` or `'View'` a simple `'if'` or `'switch'` statement can ensure that the associated allocation contains the correct `'Type'` of data structure.

2. When using `'Walk'` specific data structures could be easily identified and processed by simply examining their `'Type'` fields. Consequently, the

`'Walk'` function can easily become a powerful tool for carrying out global checks or modifications to a database.

3. If it became necessary to add new members to an existing structure or change its format simply adding a new `'Type'` value would allow both the old and new data formats to co-exist within the same file. If necessary, a simple program could be written to convert the old format into the new format. Alternatively, a statement like `'switch'` could permit multiple formats to co-exist within the file be processed appropriately.

4. If the structure of a file became corrupted the `'Type'` field typically permits the file to be easily rebuilt as the `'Walk'` function can be used to pull out all of the key data structures from the damaged file. These data structures could then be verified and used to build a new file.

Let's take the concept a little further. It is possible to use advanced object oriented concepts such as inheritance and multiple inheritance with Rockall-DB. The main restriction is that such objects cannot contain any memory pointers when used in conjunction with a `'DATABASE_HEAP'` (i.e. no memory addresses or v-tables). Consequently, newer or more complex data structures can inherit from older or less complex base classes and still be transactionally stored within a file. This provides a natural and powerful way to extend existing classes and data structures.

In summary, a few simple architectural changes allow Rockall-DB to offer many of the advantages of traditional transactional databases but with little of the associated impact on performance.

8.1.3. Understanding the Cost Metrics

It is important to understand the cost metrics associated with each of the Rockall-DB heaps. Consequently, we will consider the `'Book'` example discussed in section 7.2.1 and see how changing the Rockall-DB heap and the string type impacts performance. The precise details of the example are not important to our discussion, except to note that the example builds an index of around 12,557 unique words from a total of around 789,638 non-unique words.

Let's consider the time to execute the example for each of the Rockall-DB heaps using the `'FIXED_STRING'` class in 'Table-08-01' below:

Heap	Release Build	Debug Build
`DATABASE_HEAP`	12.02 secs	86.02 secs
`TRANSACTIONAL_HEAP`	5.02 secs	46.02 secs
`MULTI_THREADED_HEAP`	2.27 secs	14.41 secs
`SINGLE_THREADED_HEAP`	2.01 secs	12.01 secs

Table-08-01

Now, we see that for a release build of the `'DATABASE_HEAP'` that it takes 12.02 secs to process the 789,638 words, which is around 65,694 words per second. The processing for each word involves at least one index lookup and an insert into a `'SET'` or a `'TREE'`. Arguably, this is not unreasonable level of performance for a transactional database.

Nonetheless, if we substitute the `'DATABASE_HEAP'` for a `'TRANSACTIONAL_HEAP'` the elapse time drops from 12.02 secs to 5.02 secs, which is about 2.4 times faster. The change in performance is because a `'TRANSACTIONAL_HEAP'` does not need to do any file management and or support a file cache. Instead, all of its data is held in main memory.

Again, if we substitute the `'TRANSACTIONAL_HEAP'` with a `'MULTI_THREADED_HEAP'` the elapse time drops from 5.02 secs to 2.27 secs, which is about 2.2 times faster. In this case, the change in performance is because a `'MULTI_THREADED_HEAP'` does not need to create any transactions, call the lock manager, copy any data for calls to `'Update'` or clean up at the end of a transaction.

Finally, if we substitute the `'MULTI_THREADED_HEAP'` with a `'SINGLE_THREADED_HEAP'` the elapse time drops from 2.27 secs to 2.01 secs, which is about 12% faster. In this case, the change in performance is because a `'SINGLE_THREADED_HEAP'` does not need claim any locks.

What these metrics make clear is that there is a powerful performance argument for using a `'SINGLE_THREADED_HEAP'` or a `'MULTI_THREADED_HEAP'` in preference to a `'TRANSACTIONAL_HEAP'` or a `'DATABASE_HEAP'` where this is appropriate.

Let's now consider the same example but this time substitue the `'FIXED_STRING'` class with the `'FLEXIBLE_STRING'` class. The corresponding results are listed in 'Table-08-02' below:

Heap	Release Build	Debug Build
`DATABASE_HEAP`	85.16 secs	686.09 secs
`TRANSACTIONAL_HEAP`	34.10 secs	341.07 secs
`MULTI_THREADED_HEAP`	2.27 secs	15.02 secs
`SINGLE_THREADED_HEAP`	2.01 secs	14.01 secs

Table-08-02

What we see in this case is a significant drop in performance for `'DATABASE_HEAP'` and `'TRANSACTIONAL_HEAP'` but only a modest change in performance for `'MULTI_THREADED_HEAP'` and `'SINGLE_THREADED_HEAP'`. In this case, the change in performance is because a `'FLEXIBLE_STRING'` is split into two parts and so requires a large number of extra allocations (i.e. calls to `'New'`). Furthermore, this change in the number of allocations also significantly increases the number of calls related functions like `'ExclusiveView'`, `'Update'` and `'View'`. All of these are expensive calls in a `'DATABASE_HEAP'` or `'TRANSACTIONAL_HEAP'` but are typically optimized away in a `'MULTI_THREADED_HEAP'` and `'SINGLE_THREADED_HEAP'`.

Clearly, this is an extreme example but hopefully demonstrates that splitting data structures seriously hurts performance in many situations. In some regards, this is obvious as any transactional system will clearly need to record more changes in the log, do more calls to the lock manager, lock and unlock more areas of memory and possibly initiate additional file transfers for the related data.

Holding this thought for a moment, we see that relational databases and third normal form tend to drive software architectures towards larger numbers of tables and shorter row lengths. Adding insult to injury, it is not uncommon to have to join these tables together again to undo this architectural separation. This typically consumes noticeable amounts of additional CPU time. In short, it is probably prudent to try to avoid these types of structures in Rockall-DB or any other transactional database for that matter.

Let's take a look what we can do about this in Rockall-DB by considering the data structure shown in 'Example-08-01' below:

```
#include "DatabaseHeap.hpp"
#include "FlexibleString.hpp"
#include "NoLock.hpp"

typedef FLEXIBLE_STRING<DATABASE_HEAP,NO_LOCK> STRING;

typedef struct
  {
  STRING    HouseName;
  STRING    Street;
  STRING    District;
  STRING    City;
  STRING    County;
  STRING    Country;
  }
ADDRESS;
```

Example-08-01

What we see in 'Example-08-01' may seem like a very reasonable data structure for a transactional database. However, we should note that a `'FLEXIBLE_STRING'` has a fixed header part and a flexible text part. Consequently, this is not just a simple `'struct'` as it might appear but rather a `'struct'` accompanied by up to 6 additional flexible text parts (i.e. up to 7 allocations in all). Now, we could try to mitigate this by merging some of the strings as shown in 'Example-08-02' below:

```
#include "DatabaseHeap.hpp"
#include "FlexibleString.hpp"
#include "NoLock.hpp"

typedef FLEXIBLE_STRING<DATABASE_HEAP,NO_LOCK> STRING;

typedef struct
  {
  STRING    HouseAndStreet;
  STRING    DistrictAndCity;
  STRING    CountyAndCountry;
  }
ADDRESS;
```

Example-08-02

Now while this might improve performance by reducing the number of flexible text parts (i.e. from 7 to 4 allocations) it introduces other complications.

An alternative might be to replace the `'FLEXIBLE_STRING'` class with the `'FIXED_STRING'` class as shown in 'Example-08-03' below:

```
#include "DatabaseHeap.hpp"
#include "FixedString.hpp"
#include "NoLock.hpp"

typedef FIXED_STRING<DATABASE_HEAP,NO_LOCK,32> STRING;

typedef struct
  {
  STRING    HouseName;
  STRING    Street;
  STRING    District;
  STRING    City;
  STRING    County;
  STRING    Country;
  }
ADDRESS;
```

Example-08-03

Now, 'Example-08-03' would be far more efficient than 'Example-08-01' because the `'struct'` is now a fixed size and so just a single allocation (i.e. there are no additional flexible string parts). Let's now assume that a customer record consists of a name, age, date of birth, member ID and up to 3 addresses. We could code this as shown in 'Example-08-04' below:

```
#include "DatabaseHeap.hpp"
#include "FixedString.hpp"
#include "NoLock.hpp"

typedef FIXED_STRING<DATABASE_HEAP,NO_LOCK,32> STRING;

typedef struct
  {
  STRING    HouseName;
  STRING    Street;
  STRING    District;
  STRING    City;
  STRING    County;
  STRING    Country;
  }
ADDRESS;

typedef struct
  {
  STRING    Forename;
  STRING    Surname;

  ADDRESS   Address[3];

  int       Age;
  STRING    DateOfBirth;
  int       MemberID;
  }
CUSTOMER;
```

Example-08-04

Again, this is a fixed size simple allocation and so is highly efficient. Now, let's assume that for some reason we would like to store an number of the customers together in a single array. We could code this as shown in 'Example-08-05' below:

```
#include "DatabaseHeap.hpp"
#include "FixedString.hpp"
#include "NoLock.hpp"

typedef FIXED_STRING<DATABASE_HEAP,NO_LOCK,32> STRING;

typedef struct
  {
  STRING    HouseName;
  STRING    Street;
  STRING    District;
  STRING    City;
  STRING    County;
  STRING    Country;
  }
ADDRESS;

typedef struct
  {
  STRING    Forename;
  STRING    Surname;

  ADDRESS   Address[3];

  int       Age;
  STRING    DateOfBirth;
  int       MemberID;
  }
CUSTOMER;

typedef struct
  {
  int       Count;

  CUSTOMER  Customer[1];
  }
CUSTOMERS;
```

Example-08-05

Now, in this case we could create a suitably sized `'CUSTOMERS'` structure by calling `'New'`. Then we could set `'Count'` to the correct size and finally copy in all the `'CUSTOMER'` details into the new data structure. We could adjust the size of the `'CUSTOMERS'` structure at any time by calling `'Resize'` and inserting or removing `'CUSTOMER'` details as necessary.

What we are highlighting here is that the `'CUSTOMERS'` structure is still a fixed size simple allocation and so could be loaded by a single call to `'ExclusiveView'`, `'Update'` or `'View'`. A typical relational database typically requires multiple tables, multiple rows and possibly multiple joins to compute such a table. Consequently, using a structure like this in Rockall-DB would *simply blow away most relational databases in terms of performance*.

A number of the benefits of using Rockall-DB exist not because it is clever but rather because it is flexible. This flexibility allows very different transactional data structures to be built. It is not that Rockall-DB doesn't support the classical transactional data structures (i.e. such as classical relational data structures). It is because these structures are *optional* and can be used when needed rather than being forced upon developers.

In summary, when using Rockall-DB it is now possible to break the mould and think in new ways. There is now no longer any need to be shackled to third normal form. In many respects, a new day has dawned where developers can have genuine flexibility in their choice of transactional data structures. A day where words like arrays, hash tables, queues, sets, stacks and unions can be used in the same sentence as words like columns, rows, tables and transactions.

8.2. Performance

A number of performance related topics are covered in this section. These topics are presented in alphabetical order and are not necessily closely related.

8.2.1 Asynchronous Read-Ahead

A temptation in Rockall-DB is to code algorithms in a sequential style rather than a '`SET`' based style.

Let's take a look at some code that tries to sum the '`Age`' values from the structures outlined in 'Example-08-04' as shown in 'Example-08-06' below:

```
#include "DatabaseHeap.hpp"
#include "Example-08-04.hpp"
#include "RockallTypes.hpp"

int SumAges
    (
    FILE_ADDRESS    Array[],
    DATABASE_HEAP   *Heap,
    int             Size
    )
  {
  int Total = 0;
  int Count;

  for ( Count=0;Count < Size;Count ++ )
    {
    CUSTOMER *Customer;

    if ( Heap -> View( Array[ Count ],((VOID**) & Customer) ) )
      {
      Total += Customer -> Age;

      Heap -> EndView( Array[ Count ] );
      }
    }

  return Total;
  }
```

Example-08-06

The function ‘`SumAges`’ in ‘Example-08-06’ requires an ‘`Array`’ of ‘`FILE_ADDRESS`’ values (which refer to ‘`CUSTOMER`’ structures), a ‘`Heap`’ and the ‘`Size`’ of the ‘`Array`’. The function sums all the ‘`Age`’ values in the related to ‘`CUSTOMER`’ structures and returns the total.

The key concern in ‘Example-08-06’ is that the call to ‘`View`’ is synchronous. Consequently, if the ‘`Array`’ contained 1,000 ‘`FILE_ADDRESS`’ values and each of these required an file transfer taking 5ms then ‘Example-08-06’ could take up to 5 secs to execute.

Now, a small addition to this code would typically dramatically improve the performance as shown in ‘Example-08-07’ below:

```
#include "DatabaseHeap.hpp"
#include "Example-08-04.hpp"
#include "RockallTypes.hpp"

int SumAges
    (
    FILE_ADDRESS    Array[],
    DATABASE_HEAP   *Heap,
    int             Size
    )
  {
  int Total = 0;
  int Count;

  for ( Count=0;Count < Size;Count ++ )
    { Heap -> Touch( Array[ Count ] ); }

  for ( Count=0;Count < Size;Count ++ )
    {
    CUSTOMER *Customer;

    if ( Heap -> View( Array[ Count ],((VOID**) & Customer) ) )
      {
      Total += Customer -> Age;

      Heap -> EndView( Array[ Count ] );
      }
    }

  return Total;
  }
```

Example-08-07

The key change in ‘Example-08-07’ is to call ‘`Touch`’ (see section 5.4.8) for every value in the ‘`Array`’ before the first call to ‘`View`’. The call to ‘`Touch`’ will initiate all of the file transfers in the ‘`Array`’ immediately reducing the execution time of ‘Example-08-07’ from around 5 secs to under 1 sec (assuming suitable I/O hardware is available).

The Rockall-DB library contains the ‘`ROW_SET`’ class (see section 6.2.6) to assist in precisely these types of situations.

8.2.2. Automatic Allocation Alignment

All calls to `'New'` or `'Resize'` with a size that is a power of 2 and less than 4,096 will produce a memory allocation that is aligned on the same power of 2 boundary. Consequently, if the size passed to `'New'` is 1,024 then the memory allocated will always be on at least a 1,024 byte boundary.

All calls to `'New'` or `'Resize'` with a size that is a power of 2 and greater than 4,096 will produce a memory allocation that is aligned on a 4,096 byte boundary. So, if the size passed to `'New'` is 8,192 then the memory allocated will always be on at least a 4,096 byte boundary.

Consequently, there is seldom a need to any memory alignment on Rockall-DB memory allocations. Moreover, this is part of a number of a subtle optimizations which ensure that Rockall-DB allocations naturally fall into cache-lines and operating system pages to help boost performance and minimize file requests.

8.2.3. Data Density

All Rockall-DB memory allocations are packed togther nose to tail within a page with no intervening heap data structures. This dense packing of user allocations typically results in fewer cache-line misses and page faults and so leads to higher levels of performance. Additionally, Rockall-DB actively tries to allocate the lowest suitable memory address available, so as compress user allocations over time. The only downside of this approach is that over-running the end of a memory allocation in Rockall-DB will most lightly damage user data structures rather than heap data structures. Regardless, this is a bug and will need to be fixed approriately.

8.2.4. Fragmentation

All Rockall-DB heaps actively try to compress themseleves by returning the lowest memory addresses available for new allocations (i.e. as mentioned in section 8.2.3 above). Consequently, it is somewhat uncommon for Rockall-DB heaps to become significantly fragmented. However, as a memory allocation can not be moved once it has been made it is not uncomon for a small number of allocations to have high memory addresses. Typically, this is not a concern except for a `'DATABASE_HEAP'` where such allocations could prevent the compression of the associated file.

8.2.5. Storage Optimizations

All Rockall-DB `'DATABASE_HEAP'` files have a natural stride of 64k bytes. Consequently, it is optimal to select an alignment size, block size and stripe size of 64k for any related storage systems. There are cases where SAN providers and others have made alternative recommendations. Such recommendations *should typically be ignored* in relation to Rockall-DB unless supported up by actual performance measurements.

8.3. Portability

The original development for Rockall-DB was carried out on Microsoft Windows. Nonetheless, all of the operating systems interfaces are well isolated and porting to alternative operating systems is straight-forward if similar facilities to `'Kernel.dll'` are available (i.e. asynchronous I/O, files, memory space, semaphores, processes, threads and thread local store). Furthermore, Rockall-DB is also suitable for use within hardware devices or operating systems as the main file system.

Anyone interested in these areas should make direct contact with Rockall Software Ltd to discuss a source code license.

8.4. Reliability

A number of reliabilty related topics are covered in this section. These topics are presented in alphabetical order and are not necessily closely related.

8.4.1. Backups

A Rockall-DB `‘DATABASE_HEAP’` file is normal file and can be backed up in the usual way when not in use. If it is in use, it may be backed up if a snapshot of the file can be taken at some instant in time (i.e. say by a SAN). Such a backup may contain incomplete transactions and so may need to be recovered by opening it with `‘OpenFile’` with `‘ReadOnly’` set to `‘False’` to enable any necessary recovery to take place (see section 5.4.2). Alternatively, the active program could internally call the `‘CreateSnapshotFile’` function (see section 5.4.2) and dynamically create a new snapshot which could then be optionally recovered and backed up.

8.4.2. Maintenance

A Rockall-DB `‘DATABASE_HEAP’` file should not require any maintenance. Nonetheless, a number of situations may occur where it could be necessary to rebuild a `‘DATABASE_HEAP’` file after it has been corrupted, damaged (i.e. as the result of a virus) or as part of an upgrade. Consequently, it is suggested that writing a small program capable of reading any existing Rockall-DB `‘DATABASE_HEAP’` files, doing basic checks and then building a new copy of the file is a prudent investment. A number of features have been included in Rockall-DB to make this easy, such as the `‘Walk’` function (see section 5.2.9).

8.4.3. Replication

A ‘live’ replica of a Rockall-DB `‘DATABASE_HEAP’` file can be created by calling `‘CreateShadowFile’` (see section 5.4.1). A ‘live’ replica of this type cannot be used while it is being kept up-to-date and should be considered to be more of a ‘live’ backup. If it becomes necessary to make use of such a copy then no special treatment is necessary. The copy can be simply opened and used in the usual way.

8.4.4. Security

A Rockall-DB `‘DATABASE_HEAP’` file is normal file and so is subject to the usual operating system security. There are no additional levels of security except for the option of supplying encryption functions to `‘CreateFile’` and `‘OpenFile’` (see sections 5.3.1 and 5.4.3). If these functions are supplied then the contents of any related files will remain encrypted when they are not in use. However, the contents of these files will be decrypted when they are bought into main memory by Rockall-DB and will remain this way until they are removed from main memory.

8.4.5. Torn Writes

A very nasty issue with many transactional databases is the potential for a ‘torn write’. This can occur if a write (i.e. let’s say of 8k) is interupted by a power failure. A situation can arise where the first 4k of an 8k write could been written but the last 4k is *not* written. Clearly, this can easily corrupt a database and so is potentially a very serious situation.

A ‘torn write’ is automatically recovered by Rockall-DB without any intervention the next time the file is opened by `‘OpenFile’` with `‘ReadOnly’` set to `‘False’` (see section 5.4.1). No special hardware or extra steps are required beyond the requirement that the file remains readable and updateable.

8.4.6 Security

The security of transactional databases has historically been somewhat challenging. A good approach is to place all master databases behind one or more firewalls and close all the in-bound ports. Consequently, all in-bound access to the master databases would be blocked making a direct attack very difficult. In this model, any updates would need to be performed by the master database reaching out through the firewall to a less secure duplicate copy of the database which would updated in the usual way. Clearly, any updates extracted by the master database using this mechanism would need to be carefully checked before being applied. These duplicate databases could be regularly refreshed from the master copies (i.e. daily) to ensure they remained accurate and up-to-date.

While such an approach would certainly help to protect all the master databases the duplicate copies would still remain vunerable to direct attack. A number of approaches can be taken to fend off such attacks. A simple solution is to simply encrypt all the duplicate databases. However, if this encryption were cracked then an attacker would have access to all of the related data. Consequently, a better solution would be to store a small database for every client within a larger database (i.e. nested databases) and encrypt each of the smaller databases with a different encryption keys for each customer. Consequently, cracking a single encryption key would only give access to a single customers data while the remainder of the larger database would remain secure.

All of the suggestions outlined above can be implemented using Rockall-DB. These suggestions are merely a small sample of the ways security can be improved using Rockall-DB and this is not an exhaustive list of the possibilities.