CodeBase 6.0 ™
User’s Guide

The C++ Engine For Database Management
Clipper Compatible
dBASE Compatible
FoxPro Compatible

Sequiter Software Inc.

© Copyright Sequiter Software Inc., 1988-1996. All rights reserved.

No part of this publication may be reproduced, or transmitted in any form or by any means without the written permission of Sequiter Software Inc. The software described by this publication is furnished under a license agreement, and may be used or copied only in accordance with the terms of that agreement.

The manual and associated software is sold with no warranties, either expressed or implied, regarding its merchantability or fitness for any particular purpose. The information in this manual is subject to change without notice and does not represent a commitment on the part of Sequiter Software Inc.

CodeBase™ and CodeReporter™ are trademarks of Sequiter Software Inc.

Borland C++® is a registered trademark of Borland International.

Clipper® is a registered trademark of Computer Associates International Inc.

FoxPro® is a registered trademark of Microsoft Corporation.

Microsoft Visual C++® is a registered trademark of Microsoft Corporation.

Microsoft Windows® is a registered trademark of Microsoft Corporation.

OS/2® is a registered trademark of International Business Machines Corporation.

Contents

Introduction

CodeBase is a high performance database engine for C, C++, Delphi and Visual Basic programmers.

CodeBase is compatible with dBASE IV, FoxPro and Clipper file formats. CodeBase allows you to create, access or change their data, index or memo files. Applications written in CodeBase can seamlessly share files with concurrently running dBASE IV, FoxPro and Clipper applications.

CodeBase has four different interfaces depending on programming language. CodeBase supports the C++, C, Visual Basic languages and Pascal under Delphi. In addition, these interfaces can be used under a variety of operating systems, including DOS, Windows 3.1/95/NT or OS/2 and UNIX, depending on the language.

CodeBase is scalable to your needs, which means that it can be used in a stand-alone, multi-user or client-server configuration.

CodeBase has many useful features including a full set of query and relation functions. These functions use Query Optimization to greatly reduce the time it takes to perform queries and generate reports. CodeBase also has logging and transaction capabilities, which allow you to control the integrity of a database easily.

This *User's Guide* discusses the CodeBase C++ API (Application Program Interface). The CodeBase C++ API allows the programmer to write applications using CodeBase in C++. It is necessary to have a C++ compiler and some knowledge of the C++ programming language.

Refer to the *Getting Started* booklet for information on installing CodeBase, running the example programs and contacting Sequiter Software.

How To Use This Manual

This manual was designed for both novice and veteran database programmers. The *CodeBase 6.0 Getting Started* booklet details how to compile and link an example CodeBase application. This booklet is recommend reading for all users of CodeBase. The "Database Concepts" chapter of this guide discusses database terminology. Even if you are an expert database programmer, you should at least skim this chapter because some of the terms, such as tags and indexes, may be used differently than you expect. The remaining chapters deal with the basics of writing database applications with the CodeBase library. For your convenience, all example programs illustrated in this manual can be found on disk.

Manual Conventions

This manual uses several conventions to distinguish between the various language constructs. These conventions are listed below:

CodeBase classes, functions, structures, constants, and defines are always shown highlighted as follows:

e.g. **Data4::create , Code4, r4eof, E4DEBUG**

dBASE expressions are always contained by two double quotes (" ").

e.g. " 'Hello There ' + FIRST_NAME"

You should note that the containing quotes are part of the C/C++ syntax rather than part of the dBASE expression.

dBASE functions are displayed in upper case and are denoted by a trailing set of parenthesis.

e.g. UPPER() , STR(), DELETED()

C++ language functions and constructs are shown in bold typeface.

e.g. **main , for, scanf, cout, memcpy.**

C++ data types are shown bolded and encapsulated in parenthesis.

e.g. **(int), (char *),(long)**

1 Database Concepts

If you are not familiar with the concept of a *database*, simply think of it as a collection of information which has been organized in a logical manner. The most common example of a database is a telephone directory. It contains the names, phone numbers, and addresses of thousands of people. Each listing in the phone book corresponds to one *record*, and each piece of information in the record corresponds to one *field*. The phone book excerpt below illustrates this concept.

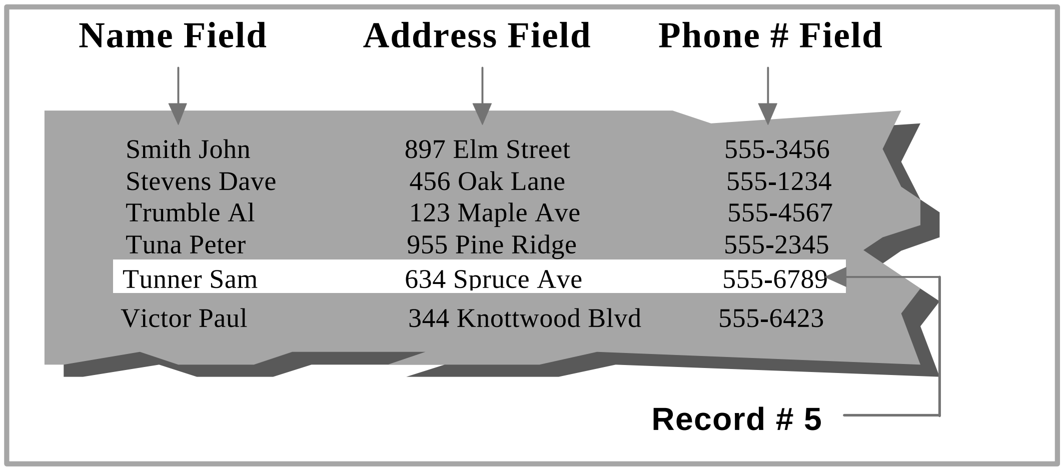


Figure 1.1 Phone Book Example

As shown in this example, each *data file* (also known as a table) is a collection of one or more fields (also known as a tuple). Each of these fields has a set of characteristics that determine the size and type of data to be stored. Collectively, these field descriptions make up the structure of your data file.

CodeBase File Format

CodeBase uses the industry standard .DBF data files. This standard allows for several types of fixed width fields and one type of variable length field.

Fields

The **.DBF** standard uses four attributes to describe each field. These attributes are Name, Type, Length, and Decimal. They are listed below. For detailed information on field attributes, please refer to the **"Field4info"** class chapter in the CodeBase *Reference Guide*.

- **Name**: This simply refers to the name that will be used to identify the field. Each field name can be a maximum of 10 characters, and each field name must be unique within a data file and must consist of alphanumeric or underscore characters. The first character of the name must be a letter.
- **Type**: The type of the field determines what kind of information should be stored in the field. There are eight different types that can be specified for any given field. They are Character, Numeric, Floating Point, Date, Logical, Memo, Binary and General.
- **Length**: This attribute refers to the number of characters or digits that can be stored in the field.
- **Decimal**: This attribute applies only to Numeric and Floating Point fields. It specifies the number of digits after the decimal point.

Records

A record consists of one instance of every field, and has a unique *record number* and a *deletion flag* associated with it. The record number indicates the physical position in the data file while the deletion flag is used during the deletion process. The deletion flag is set to true (non-zero) to indicate that the record should be removed from the data file when the data file is packed.

Tags

A *tag* determines the ordering in which the records in a data file are presented. The tag ordering does not affect the physical ordering of the records in the data file, only the order in which they can be accessed. The information that this ordering is based upon is called the *index key*.

In the phone book example, the index key is based on the Name field and the actual index keys for records 1 to 3 are "Smith John", "Stevens Dave", and "Trumble Al". If you wanted to display the phone information sorted by the phone number, you would create a tag based on the phone number field.

Indexes

An *index* is a file containing the sorted index keys for one or more tags. There are several formats of index files which CodeBase supports.

- "**.MDX**" (dBASE IV)
- "**.NTX**" (Clipper)
- "**.CDX**" (FoxPro)

The "**.CDX**" and "**.MDX**" allow you to have multiple tags in each index file, and to have *production indexes*. A production index is an index file that is opened automatically when the associated data file is opened. The "**.NTX**" format limits you to one tag per index file, and does not allow for production indexes. CodeBase does, however, provide this functionality through the use of group files. Refer to the "Indexing" chapter for more details on index file formats.

Filters

Filters are used to obtain a subset of the available data file records. The subset is based on true/false conditions calculated from data file field information. Only records that pass through the filter have entries in the tag. This allows for fast access to a data subset. Refer to the "Indexing" chapter in this guide and the **"Tag4info"** class chapter in the CodeBase *Reference Guide* for details on using filters.

Relations

A relation is a connection between two or more data files. A relation determines how records in related data files can be found from a record in the current data file. Refer to the "Queries And Relations" chapter for more details on relations.

2 C++ Programming

The C++ programming languages are perhaps the most popular and flexible languages currently in use. While it is not the purpose of this manual to teach you how to program in C++, this chapter deals with some of the common difficulties encountered by C++ programmers when first using the CodeBase library.

Although CodeBase is straight forward and easy to use, it does require a basic understanding of the C++ language, including the use of structures and pointers. The following sections illustrate some common problems and their remedies.

Memory Corruption

One of the main features of C++ is the flexibility in using memory. Unfortunately this flexibility also lets you change memory that you shouldn't causing memory corruption. Memory corruption, due to programming oversights, causes the most difficult bugs in C++ programs.

The sizeof Operator

Many situations that cause memory corruption can be avoided by using the C++ language's **sizeof** operator. One of the biggest culprits of memory corruption occurs when data is written beyond the end of a string character array. The following code fragment illustrates this error:

INCORRECT

```
char temp[8];
strcpy(temp,"This string is longer than 8");
```

This sort of problem can be avoided by using the **sizeof** operator and a string copy function that copies a maximum number of characters, such as **strncpy** or **memset**.

CORRECT

```
char temp[8];
memset(temp,'X',sizeof(temp));
```

The **sizeof** operator may also be used with some **Str4** derived classes to ensure that only allocated memory is used. Once set, the **Str4** derived classes ensure memory is not overwritten.

```
char buf[8] ;
Str4len strBuf( buf, sizeof(buf) ) ;

strBuf.set( 'X' ) ;
cout << strBuf.str( ) << endl ; // output 8 'X's

strBuf.assign( "This string is longer than 8" ) ;
cout << strBuf.str( ) << endl ; // outputs 'This str'
```

Memory Scoping

Another common mistake is dealing with memory scoping. In the next example, the programmer wants to return a string that has been filled with data.

INCORRECT

```
#include "d4all.hpp"

char *SetString( void )
{
    char buf[ 15 ];
    memcpy( buf, "HELLO WORLD!", 13 ) ;
    return buf ;
}

void main( void )
{
    char *Message;
    Message = SetString( );

    ... more stuff ...
}
```

This program may appear to work at first, but the memory for the variable *buf* is automatically deallocated after the *SetString* function is exited. Since the memory to which *Message* points is free, it can be reused by the system at any time. Consequently, using the information pointed to by *Message* is an error.

To avoid this situation, several correct methods can be used. The first uses static memory.   Static memory is not deallocated while the program is running, and the same memory is used every time the function is called.

CORRECT

```
#include "d4all.hpp"

char *SetString( void )
{
    static char buf[ 15 ] ;
    memcpy( buf, "HELLO WORLD!", 13 ) ;
    return buf ;
}

void main( void )
{
    char *Message;
    Message = SetString();

    ... more stuff ...
}
```

The next example accomplishes the same result by allocating the memory in the calling function by declaring the variable *message* in **main**.

CORRECT

```
#include "d4all.hpp"

void SetString( char *buf, int size)
{
    memcpy( buf, "HELLO WORLD!", size < 13 ? size : 13);
}

void main( void )
{
    char Message[ 15 ];
    SetString( Message, sizeof( Message ) );

    ... more stuff ...
}
```

Finally you can dynamically allocate the memory. Remember to deallocate it when you are done.

CORRECT

```
char *SetString( void )
{
    char *buf = new char[ 15 ] ;
    memcpy( buf, "HELLO WORLD!", 13 ) ;
    return buf ;
}

void main( void )
{
    char *Message;
     Message = SetString( );

    ... more stuff ...

    delete Message ;
}
```

Structure Parameters

Many of the common problems encountered by novice C++ users deal with the passing and returning of structure parameters.

Structure passing

Consider this example. The programmer wishes to pass a structure to a function that initializes several of its flags.

INCORRECT

```
struct myStructSt
{
    int buttonDown ;
    int clipping ;
} MY_STRUCT ;

void InitFlags(MY_STRUCT ms)
{
    /*set some flags */
    ms.buttonDown = 0 ;
    ms.clipping = 0 ;
}

void main( void )
{
    MY_STRUCT myStruct ;
    myStruct.buttonDown = myStruct.clipping = 1 ;
    InitFlags( myStruct ) ; // reset to zero ??
    //  buttonDown and clipping are still set to 1 !!!
}
```

When you pass a variable (in this case a structure) as a parameter, the function actually receives a copy of that variable. Therefore any changes which are made to the structure members in the function, are lost when the function returns.

This causes several problems. First of all, memory is wasted by storing a copy of the structure members. Any changes that are made to the structure members are lost.

The correct way to pass a structure to a function, is to pass it in as a pointer. This is illustrated in the corrected version:

CORRECT

```
struct myStructSt
{
    int buttonDown ;
    int clipping ;
} MY_STRUCT ;

void InitFlags(MY_STRUCT *ms)
{
    /*set some flags */
    ms->buttonDown = 0 ;
    ms->clipping = 0 ;
}

void main( void )
{
    MY_STRUCT myStruct ;
    myStruct.buttonDown = myStruct.clipping = 1 ;
    InitFlags( &myStruct ) ; // reset to zero
    //  buttonDown and clipping are now zero
}
```

Class Object Passing

When passing class objects to functions, the same rules apply: passing an object creates a copy, passing a pointer (or reference) to an object allows changes made to stay in effect. Many CodeBase structures (such as the **Code4** class) have many member variables and maintain several sets of linked lists that can be corrupted by accessing them through a copy. Please see the "Copying CodeBase Classes" chapter in this manual for a more detailed explanation of which classes may and may not be copied.

Note

When in doubt, pass a reference.

3 Using the String Classes

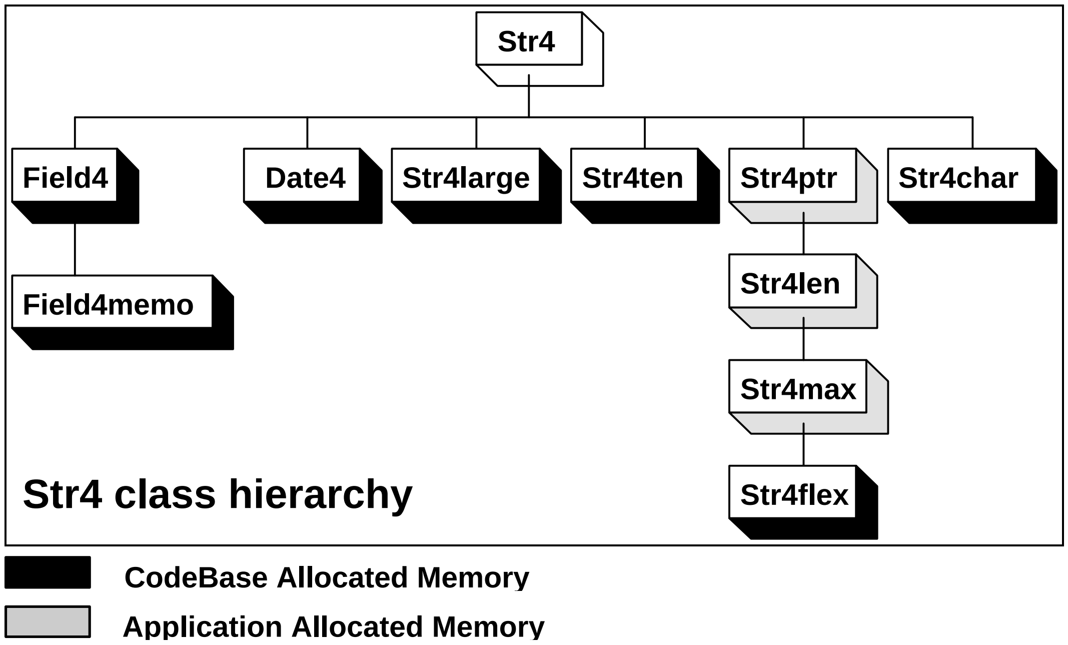


Figure 3.1 Str4 class hierarchy

The CodeBase string classes provide C++ programmers with a safe, flexible, and reliable method of storing, retrieving, and manipulating character data. They contain flexible memory management which gives the programmer maximum control, while ensuring that inappropriate memory is not overwritten.

The CodeBase string classes also contain simple, easy to remember member functions for data manipulation. Strings may be added together with **Str4::add**, inserted in one another with **Str4::insert**, and compared with standard C++ relational operators.

In addition, the string class also forms the basis for manipulating the information in data bases. **Field4**, which is derived from the virtual base class **Str4**, is used to directly manipulate the information in the record buffer. Since the string class makes use of virtual functions, data manipulation made with the base class functions automatically call **Field4::changed** to ensure that CodeBase flushes the changes to disk at the appropriate time.

The string class also forms the basis for the CodeBase date class. **Date4** builds upon the functionality of the **Str4** class to provide arithmetic operators for addition and subtraction of days, as well as functions for retrieving character representations of the day and month of the date stored.

The string classes of CodeBase are organized to provide maximum flexibility and control. However, since there are quite a few of them to choose from, the novice CodeBase user may not be able to quickly select the most appropriate class for his/her needs. This chapter will assist in the correct usage of the classes and provide some examples on using the classes.

One of the major benefits of using the CodeBase string class is that they ensure that memory is not accidentally overwritten. This avoids many of the problems discussed in the previous chapter.

Memory Usage

One of the main differences between the different string classes is how the memory is allocated for new objects. Some classes have internal static memory, one dynamically allocates memory, and some require that the application provide the memory used by the object.

Most of the **Str4** derived classes take care of their own memory. This frees the programmer from having to allocate and deallocate memory for each object used in the application. For a detailed list of the exact memory usage for the string classes, see "Appendix D: CodeBase Limits" in the CodeBase *Reference Guide*. Listed below are all the **Str4** derived classes and the amount of memory available to the object.

The classes which do not have memory allocated for them by the class provide the programmer with the flexibility of the CodeBase string class member functions, while working with standard and application defined data types.

Class	Available Memory
Date4	8 characters + 1 for a null.
Field4	Memory is allocated from the heap as part of the **Data4** record buffer and is the size of the field.
Field4memo	Memory is allocated from the heap in blocks the size of the **Code4::memSizeMemo**.
Str4	N/A
Str4char	1 character.
Str4flex	Memory is allocated from the heap to accomodate the largest value assigned to the object.
Str4large	255 characters + 1 for a null.
Str4len	Memory is application allocated.
Str4max	Memory is application allocated.
Str4ptr	Memory is application allocated.
Str4ten	10 characters + 1 for a null.

Figure 3.2 Str4 class memory allocation

CodeBase Allocation

The choice between the classes that have memory allocated for them is determined by how the object is to be used.

- **Date4 -** This class performs date manipulations and may be assigned to date fields.
- **Field4memo -** Use this class when data file fields are used generically or when memo, binary or general fields are used.
- **Field4 -** Use this class when data file fields are used.
- **Str4char -** This class is used to perform string computations and calculations on single character data. If more than one character is necessary, use a larger class.

- **Str4large -** Use this class to store larger character data. However, if the application might store more than 255 characters, use the **Str4flex** class or **Str4max** and allocate your own buffer.
- **Str4ten -** This class is used to store small amounts of data. If more than 10 characters may be stored, use **Str4large** or **Str4flex**.
- **Str4flex -** This class dynamically allocates memory at run-time. As a result, it provides the most flexible method of storing data. However, since memory is allocated off the heap, in some low-memory situations, **Str4flex** will not be able to get the requested amount of storage space.

Application Allocation

When one of the classes with CodeBase memory allocation does not suffice, or when non string data is to be used, the **Str4ptr**, **Str4len**, or **Str4max** classes may be used. These classes require that the application allocate the memory that is used.

Each one of these classes build on one another to provide maximum flexibility.

- **Str4ptr** works only with null terminated strings. All **Str4** functions are fully operational in this class. The length of the object is determined by the number of characters (before the first null) of the initialization string. This class allows standard C++ character arrays to be treated as if they were CodeBase strings.
- **Str4len** works with any type of data. It provides all of the functionality of the **Str4ptr** class, but uses a specified length instead of using the length (up to the first null) of the initialization string. This means that if non-character data is used, such as a user defined structure, the **Str4len** functions will ensure that inappropriate memory is not written to.
- **Str4max** builds on the **Str4len** by providing a fixed maximum and a flexible current size. In general, this is mostly applicable to strings.

Varying the length of an object

CodeBase provides functions to efficiently and effectively store information in the string class objects. Since different classes can store different types of data, it follows that different classes have different lengths associated with them.

Class Name	Current Length	Maximum Length
Date4	Fixed at 8	Fixed at 8
Field4	Fixed at length of field	Fixed at length of field
Field4memo	Flexible	Flexible
Str4char	Fixed at 1	Fixed at 1
Str4flex	Flexible	Flexible
Str4large	Flexible	Fixed at 255
Str4len	As constructed	Fixed at Current Length
Str4max	Flexible	As constructed
Str4ptr	Fixed at the length of initialization string	Fixed at the length of initialization string
Str4ten	Flexible	Fixed at 10

Figure 3.3 Length Variations

For example, the **Str4char** class is designed to operate on a single character of data; it has a length of one. **Date4**, which stores dates in standard format ("CCYYMMDD") has a length of 8. **Field4** objects store data in database fields, and so have a length equal to the length of the field. In all these cases, the length of the object is determined by the purpose of the class and the type of information it stores.

When information is stored in a CodeBase string class, an attempt is made to adjust the length of the string object so that it matches the length of the information stored. Depending upon the purpose of the class, the length adjustment may fail because the length of an object is fixed at a pre-determined length, or may succeed because its length is flexible.

Fixed Current Length

A few of the string classes (**Date4, Field4, Str4char, and Str4len**) may not alter their length. They have a fixed length that is not altered by the storage of larger or smaller strings. Attempts made to lengthen or shorten the string (by assigning larger or smaller data to the object) fail and the following actions take place:

- When a larger string is stored, the excess information is truncated at the length of the object.
- When information that is shorter than the length of the object is stored, the remaining memory for the object is filled with spaces.
- These actions ensure that memory is not overwritten, while storing information accurately.

Example

As an example, take the case where a **Field4** object references a 20 character data base field. If "Jonathan Christopherson" (length of 23 characters) would be stored in the field, only "Jonathan Christopher" would actually be placed in the field, since the source string is three characters too big.

If "Jonathan Christopher" was already stored, but it was necessary to store "John Smith" in the field, "John Smith   " (with the trailing spaces) would actually be placed in the field. Failure to pad out the object with spaces would result in a field containing "John Smithhristopher". (The later would occur if **Str4::replace** was used instead of **Str4::assign**)

Flexible Current Length

The other CodeBase string classes (**Field4memo, Str4flex, Str4large, Str4max,** and **Str4ten**) all have a flexible current length. That is, when a string is stored within one of these objects, the length of the object is adjusted to accommodate the length of new string. When this occurs, the following rules are applied:

- When a string larger than the current length is stored, the current length is expanded to accomodate the new information. If the new length is larger than the maximum memory allocated for the object, the excess information is truncated at the maximum length of the object.
- When information that is shorter than the current length of the object is stored, the current length is adjusted to the shorter

length, and a null is placed immediately after the new information (if space allows).

The new current length is then used whenever the object's contents are retrieved.

Example

An **Str4ten** object, as an example, has 10 bytes of memory which may be used to store information. When constructed, the object contains no information, and so has a current length of zero. If "JOHN" was stored in the object, the length would be increased to four. An attempt to store "JOHN PETERSON", however, would exceed the maximum size of the object, and only "JOHN PETER" would be stored. If "NORMAN" was then stored, the length of the object would be reduced from ten to six, and a null would be placed at byte seven.

Flexible Maximum Lengths

Str4flex and **Field4memo** handle the current and maximum lengths in a slightly different manner than the other CodeBase string classes. These are the only classes that actually change the amount of memory used for the object. When a larger string is stored within a **Str4flex** or **Field4memo** object, more system memory is allocated and the current length of the object is increased to the length of the new information.

In order to streamline the memory allocation for **Str4flex** and **Field4memo** objects, CodeBase often allocates more memory from the system than is actually required to store the requested data. This memory used is not reduced when a smaller value is stored, instead the largest amount of memory used by the object is maintained in case the length is to grow again. This minimizes the overhead involved in repeated allocation and deallocation.

The memory for a **Str4flex** object is freed by **Str4flex::free** and the class destructor. The memory for the **Field4memo** object, however, is freed when the application calls **Field4memo::free** explicitly, uses **Field4memo::setLen** to set the length to zero, or when the data file is closed.

In low memory situations, it is probably a good idea to avoid using the **Str4flex** class. As an alternative, explicitly allocate the appropriate memory and use a **Str4max** object.

Copying Strings

Str4 derived classes have several functions for storing data within the object. The most used are: **Str4::assign, Str4::replace** and **Str4::insert**.

These functions change the contents of the object, ensuring that memory is not overwritten, and attempting to null terminate whenever space allows. These functions are safer than using the C function **memcpy** which may overwrite and corrupt memory when used incorrectly.

Str4::assign completely replaces the contents of the object with the value passed. If the length of the new information is greater than the current length of the object, **Str4::assign** attempts to increase the length of the object.

Listed below is an example program that uses **Str4::assign** with a number of different types of data.

```
//str1eg.cpp

#include "d4all.hpp"
extern unsigned _stklen = 10000 ; // for all Borland compilers

void main( )
{
    Code4 cb ;
    Data4 data( cb, "INFO" ) ;
    Field4 field( data, "NAME" ) ;

    Str4ptr newName( "JOHN JONES" ) ;
    data.top( ) ;
    data.lockFile( ) ;
    field.assign( newName ) ; // store JOHN JONES in record 1

    Str4large newerName( newName ) ; // copy JOHN J  ONES
    newerName.replace( Str4ptr("BILL") ) ;   // make it BILL JONES
    data.skip( ) ;
    field.assign( newerName ) ;   // store BILL JONES in record 2
    Str4large newestName( newerName ) ;  // copy BILL JONES
    newestName.insert( Str4ptr("A. "), 5 ) ;   // make it BILL A. JONES
    cb.initUndo( ) ; // close the files and free memory
}
```

Str4::assign Since the **Field4** class is derived from the **Str4** class, all of the **Str4** member functions operate on fields. The following line of code demonstrates how to change the contents of a field. The contents of *newName* are copied into the record buffer at the position of the NAME field.

```
field.assign( newName ) ; // store JOHN JONES in record 1
```

The **Str4::assign** attempts to change the size of the object, but since **Field4** hasn't overloaded **Str4::setLen**, the length of the field object remains the same. **Str4::assign** completely replaces the contents of the object, so any information already stored in the field object is replaced by 'JOHN JONES'.

In addition, **Str4::assign** calls **Field4::changed** to mark the record buffer as "changed". When appropriate, JOHN JONES is flushed to disk.

Str4::replace CodeBase provides more than "all or nothing" assignment capabilities. That is, string objects do not need to have their entire contents erased and overwritten simply to change their value. A handful of other functions, such as **Str4::replace,** provide greater flexibility.

In the following sample of code, **Str4::replace** is used to change the first four characters of the object. The rest of the *newerName* object remains unchanged.

```
Str4large newerName( newName ) ; // copy JOHN JONES
newerName.replace( Str4ptr("BILL")) ; // make it BILL JONES
```

If the replacement string is longer than the current length of the object or if it extends beyond the end of the object, the object is lengthened (if possible) and as much as possible of the replacement string is copied. This is not the case in the above code, since the *newerName* object's length at this point is 10 (the length of 'JOHN JONES').

Str4::insert One string may be placed within another without overwriting the contents of the original string by using **Str4::insert**. As the name implies, the new string is inserted into the original string and the original contents are shifted to the right. During the shifting of the object, an attempt is made to increase the size of the object to accomodate the new characters. If the maximum size of the object is exceeded, there will be some data loss from the characters shifted to the right.

```
Str4large newestName( newerName ) ;  // copy BILL JONES
// make it BILL A. JONES
newestName.insert( Str4ptr("A. "), 5 ) ;
```

If in the above example *newestName* had been a **Str4ten** object, The insertion of 'A. ' would have exceeded the maximum size of the object. 'A.' would have been inserted, but three characters of the original object would have been lost. The resulting object would have been 'BILL A. JO'.

The **Str4ptr** constructor may be used to temporarily convert a simple null terminated character array into a **Str4** derived object. Several CodeBase functions take **Str4** objects instead of **(char *)**. In these cases it is appropriate to use the **Str4ptr** constructor to simply convert a string to a **Str4** object.

Comparing Strings

Once **Str4** objects have a value they may be compared with other **Str4** objects using the comparison operators. These operators take into account both contents and length.

For the comparison sake, an object is considered equal if both the contents and the length of the fields are the same. "BOB" and "BOB " are not considered equal.

When the relational comparisons (<, >, <=, >=) are used, a **Str4** object is considered "less than" another if it is alphabetically closer to 'A'. If the lengths of the objects are different, but the contents of the fields are the same (to the shorter length), the shorter object is considered "less than" the longer.

```
//str2eg.cpp
#include "d4all.hpp"
extern unsigned _stklen = 10000 ;

void main( )
{
    Code4 cb ;
    Data4 data( cb, "INFO" ) ;
    Field4 name( data, "NAME" ) ;
    Str4large max, min ;

    data.top( ) ;
    max.assign( name ) ;
    min.assign( name ) ;

    for( data.skip( ); !data.eof( ); data.skip( ) )
    {
        if( min > name )
            min.assign( name ) ;
        if( max < name )
            max.assign( name ) ;
    }

    if( min == max )
        cout << "All records have the same name  : " << min.ptr( ) << endl ;
    else
    {
        min.trim( ) ;
        max.trim( ) ;
        cout << min.ptr( )
             << " would come first in an alphabetic listing" << endl ;
        cout << max.ptr( )
             << " would come last in an alpahbetic listing" << endl ;
    }

    data.close( ) ;
    cb.initUndo( ) ;
}
```

The above example does comparisons of the NAME field of the data file, saving the entries closest to 'Z' and closest to 'A'.

In this example, the comparison of lengths is unnecessary, since a field is always padded to its maximum size and the **Str4::assign** with the **Str4large** class always sets the length to the size of the field.

Str4 vs. char *

Standard C++ character arrays may be compared to **Str4** derived objects in a number of ways.

C++ functions

First, **Str4::str** may be called to obtain a null terminated copy of the object. This copy may then be used with standard C++ comparison functions such as **strcmp**. The return from these comparison functions can then be used to determine the exact comparison. However **Str4::str** may not be used in some operating environments. See **Str4::str** in the CodeBase *Reference Guide* for more information.

```
char buf[] = "aaa" ;
Str4ten string( "bbb") ;
if( strcmp( buf, string.str( ) ) < 0 )
{
// a negative return means param1 is smaller
    cout << buf << " is less than " << string.str( ) << endl ;
}
```

Using **Str4ptr**

A second method of comparing a character array with a **Str4** object is to promote the character array to a **Str4ptr** object and use the intuitive comparison operators. This is done by calling the **Str4ptr** constructor with the character array, without creating a permanent object.

```
char buf[] = "aaa" ;
Str4ten string( "bbb" ) ;

if( Str4ptr(buf) < string ) // buf is promoted to a Str4ptr
    cout << buf << " is less than " << string.str( ) << endl ;
```

This method constructs a temporary **Str4ptr** object which is used for the comparison, but then falls out of scope. The **Str4ptr** constructor is fast and efficient, making this method a good choice.

Using Numbers

The **Str4** class provides three conversion operators which may be used to retrieve numerical information from strings. In essence, whenever a **Str4** object needs to be used in numerical computations or comparisons, it may simply be cast to the correct type. The conversion operators are then automatically called to convert the contents of the object into a numeric value.

If the string object does not contain a character representation of a number, zero is returned. The following example shows the basics of using the numeric conversion operators with a **Str4** derived class **Field4** to retrieve ages from the data file, do comparisons with them, and increment them.

```
//str3eg.cpp
#include "d4all.hpp"
extern unsigned _stklen = 10000 ;

void main( )
{
    Code4 cb ;
    Data4 data( cb, "INFO" ) ;
    Field4 age( data, "AGE" ), name(  data, "NAME" ) ;

    // Make everyone one year older, except those people
    // who are 39 'and holding'
    data.lockFile( ) ;
    for( data.top( ); !data.eof( ); data.skip( ) )
    {
        if( 39 != int( age ))
        {
            age.assignLong( long( age ) + 1L ) ;
            cout << name.str( ) << " is now " << int( age ) << endl ;
        }
        else
            cout << name.str( ) << " is 39 and holding" << endl ;
    }
    data.close( ) ;
    cb.initUndo( ) ;
}
```

Retrieving as numbers

Once a conversion operator is used, the return value may be used just like any other numeric value. So in the following code, the standard C++ operator != is used to compare the integer 39 with the integer returned from **Str4::operator int**.

```
if( 39 != int( age ))
{
    age.assignLong( long( age ) + 1L ) ;
    cout << name.str( ) << " is now " << int( age ) << endl ;
}
```

Storing numbers

CodeBase has two functions which convert numeric values into their character represention and store them in **Str4** objects. They are **Str4::assignLong** and **Str4::assignDouble**. In addition to long and double values, these functions, through the automatic C++ conversions, handle integers and floats as well.

The above code fragment illustrates the use of **Str4::assignLong**. In order to increment the value of the field, the **Field4** object age must first be converted into a long value and have one added to it. The sum is then assigned back to the field's object.

Str4::assignLong and **Str4::assignDouble** both automatically call **Field4::changed** to mark the current record as having been changed. The changes to the AGE field are then flushed to disk at the appropriate time.

In summary, the **Str4** derived classes provide a safe and flexible method of dealing with character strings. In addition, the powerful assignment, replacement, and insertion functions make updating the strings simple.

4 DataBase Access

Now that you have been introduced to the concepts of database management, you're ready to start programming some simple database programs using C++. This chapter takes you through the details of manipulating the data files by explaining how to create and open data files, store and retrieve data, and how records are added and deleted.

Getting Started

To make the task of learning CodeBase easier, all the examples in this manual only use the standard C++ input and output functions (**cout, printf, cin, scanf**). These programs will run with any ANSI C++ compiler. Since the application programming interface (API) of CodeBase is the same for all environments, you can simply paste the example CodeBase functions into your application.

To run any of the tutorial programs, compile the program and link it to the CodeBase library according to the instructions in the *Getting Started* booklet.

CodeBase Components

In addition to the elements of standard C++ programs, every CodeBase program will contain CodeBase classes to perform the basic database management.

CodeBase Classes

All data manipulation is performed by CodeBase classes which are identified by the format of their names. These names are mixed case starting with one capital letter, followed by two to five lowercase letters and the number four. The class name describes the purpose of the class. Some common class names include **Data4** (for database management), **Date4** (for performing date manipulations, **Field4** (for accessing fields), and **File4** (for accessing files).

When a CodeBase class is derived from another class, or when its purpose is similar to another CodeBase class, it uses the name of the base class plus a descriptive suffix. For instance **Field4memo** (for accessing memo fields) is derived from **Field4**, and **File4seqWrite** is similar in purpose to **File4**.

CodeBase Constants

In addition to CodeBase classes and derived classes, you can also use CodeBase constants. Most constants are integers which are used for return values and error codes. These constants have names which usually start with **r4,** or **e4**. Some examples include **r4success**, **r4locked**, **e4memory**, and **e4open**. These constants are listed in more detail in "Appendix A: Error Codes" and "Appendix B: Return Codes" of the *Reference Guide*.

Overview

The next section introduces some the of the important CodeBase classes and examines how they are related to data files.

The **Code4** class

Foremost of the CodeBase classes is **Code4**. This class contains settings and information used by most of the other CodeBase classes.

Note

Most applications only require one **Code4** object. If you are using more than one **Code4** object, you will most likely be wasting memory resources. Exceptions to this rule are discussed later in this chapter.

The **Data4** class

The **Data4** class is used to reference a particular data file. When a data file is opened, the information for the **Data4** object is allocated and stored in the **Code4** class object. A pointer to this internally allocated information is stored in the **Data4** objects when they are constructed. When you call a function that operates on a data file, you specify which data file you want by using the **Data4** object for that data file. Note that more than one **Data4** object can reference the same open data file. However, since the information for the object is stored within the **Code4** class object, each new object constructed with **Code4::data** contains the same information, including the same current record and optimization settings.

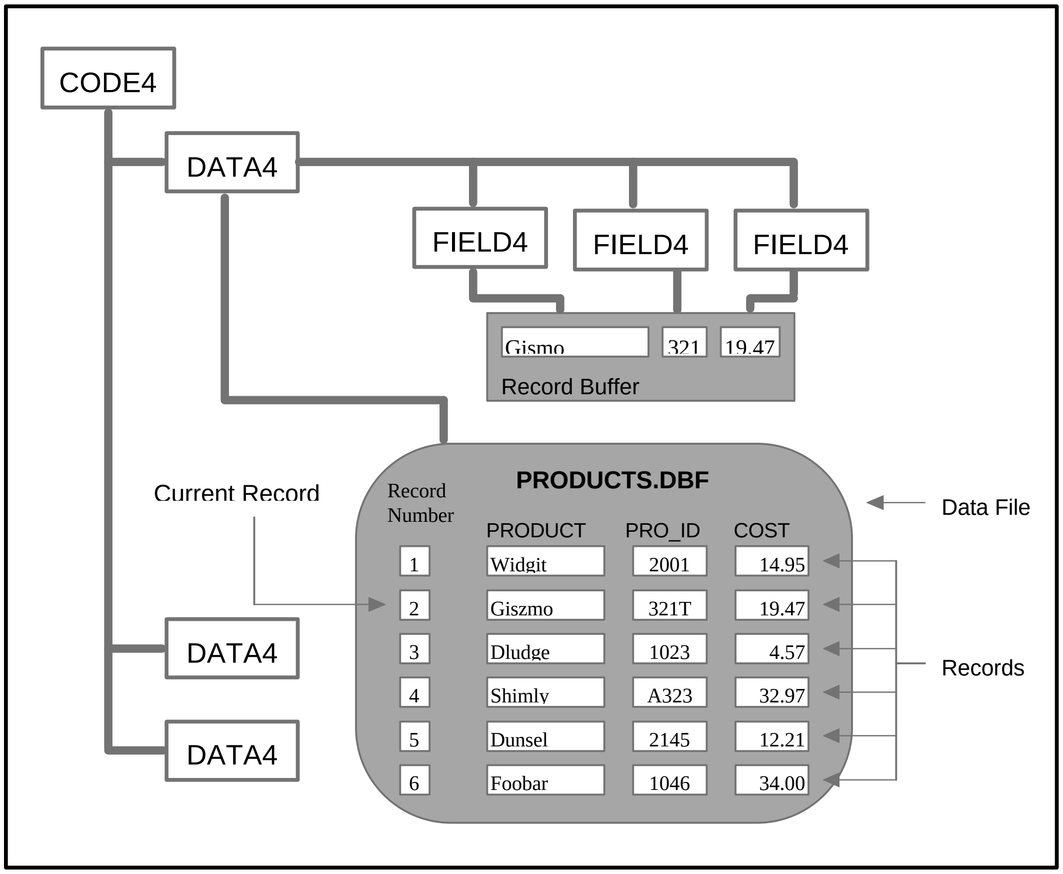


Figure 4.1 Basic CodeBase Classes

The record buffer

Each database has a memory area associated with it called the *record buffer* which is used to store one record from the data file. The record that is stored in the record buffer is called the *current record*.

Changes made to the record buffer using the **Field4** member functions are automatically written to the data file before any new current record is read.

The **Field4** class

A **Field4** object is used to reference a particular field in the record buffer. When the data file is opened, the information for the **Field4** objects is automatically allocated for each of its fields. When you construct a **Field4** object for the field you can use the member functions to modify the record buffer and change the contents of the field on disk.

An Example CodeBase Program

At this point, you are ready to write your first useful program that uses the CodeBase library. This program will display all of the records in any data file whose name you provide as a command line parameter.

PROGRAM SHOWDATA.CPP

```
#include "d4all.hpp"
extern unsigned _stklen = 10000 ;

int main(int argc,char *argv[])
{
    if(argc != 2)
    {
        cout << "USAGE: SHOWDATA <FILENAME.DBF>"<< endl;
    exit(0) ;
    }

    Code4 codeBase ;
    Data4 dataFile( codeBase, argv[1] ) ;
    codeBase.exitTest( ) ;

    int numFields = dataFile.numFields( ) ;
    for( int rc = dataFile.top( ); rc == r4success ;
                                        rc =    dataFile.skip( ))
    {
        for( int j = 1; j <= numFields; j++ )
        {
            Field4memo field( dataFile, j ) ;
            cout << field.str( ) ;
        }
        cout << endl ;
    }

    dataFile.close( ) ;
    codeBase.initUndo( ) ;
    return 0 ;
}
```

Explanation :

The next section contains a brief explanation of the SHOWDATA.CPP program, which is followed by a more detailed explanation.

Include Files

This program, like almost every program you write using the CodeBase library includes the "D4ALL.HPP" include file. This file contains the definitions and includes the type definitions for the CodeBase library. You will need to include this file in any module which uses CodeBase functions or structures. In addition, this header file includes most header files necessary for standard C++ string and output operations.

The number of arguments is checked and the program is terminated if an incorrect number was provided.

```
if(argc != 2)
{
    cout << "USAGE: SHOWDATA <FILENAME.DBF>"<< endl;
    exit(0) ;
}
```

Class objects

In addition to some variables of standard types, this program uses three CodeBase classes: **Code4**, **Data4** and **Field4memo**.

```
Code4 codeBase ;
Data4 dataFile( codeBase, argv[1] ) ;
 ...
Field4memo field( dataFile, j ) ;
```

The **Code4** object is automatically initialized as it is constructed. The **Data4** constructor is used to open the data file specified by the command line argument.

Code4::exitTest is called to terminate the program if an error has occurred while constructing the **Code4** object or opening the command line data file.

```
codeBase.exitTest( ) ;
```

The number of fields in the data file is obtained by a call to **Data4::numFields**.

```
numFields = dataFile.numFields() ;
```

Next, a **for** loop is used to load each record into the record buffer. This **for** loop consists of the following operations: a call to **Data4::top** to load the first record, a call to **Data4::skip** to load the next record, and a check to see if **Data4::top** or **Data4::skip** have reached the end of the file.

```
int numFields = dataFile.numFields( ) ;
for( int rc = dataFile.top( ); rc == r4success ;
                                            rc = dataFile.skip( ))
{
```

Inside this **for** loop, a second **for** loop is used to iterate through the fields by incrementing a counter from one to the total number of fields in the data file.

```
for( int j = 1; j <= numFields; j++ )
{
```

This counter is used in conjunction with the **Field4memo** constructor to construct an object for the specific field. A null terminated pointer to the contents of that field is returned from a call to **Field4memo::str**.

```
    Field4memo field( dataFile, j ) ;
    cout << field.str( ) ;
}
```

The data file is closed by the **Data4::close** function, **Code4** is uninitialized, and the program is exited.

```
    dataFile.close( ) ;
    codeBase.initUndo( ) ;
    return 0 ;
}
```

Once a **Code4** object is constructed and initialized, all of the flags and member variables are set to their default values. Normally only one **Code4** object should be initialized per application.

After it has been initialized, you can then change the value of the **Code4** object's flags.

Opening Data Files

Only after a **Code4** object has been initialized, can data files be opened with **Data4::Data4** or **Data4::open**.

Code fragment from SHOWDATA.CPP

```
Data4 dataFile( codeBase,argv[1]) ;
codeBase.exitTest( ) ;
```

Data4::open and the **Data4::Data4** constructor accept a string containing the path and file name of a data file. If the specified file name does not contain an extension, the default extension of "**.DBF**" is used. If no path is specified, the current directory is searched.

Constructing a **Data4** object

If the file is located, it is opened and the information for the file is used to construct a **Data4** object.

If the file does not exist, or if CodeBase detects any other error, an error message is displayed and an invalid **Data4** object is constructed.

Checking for errors

Code4::exitTest checks for any such errors and terminates the program if one has occurred. Alternatively, if you did not want to exit the program, you can call **Data4::isValid** to determine if the file was successfully opened.

After the data file has been successfully opened, the current record number is set to '-1' to indicate that a record has not yet been read into the record buffer.

Closing Data Files

Since **Data4** does not have a destructor, it is important that any open data files are closed. This ensures that any changes to the data file are updated correctly.

There are two functions that you can use. They are **Data4::close** and **Code4::closeAll**. **Data4::close** closes the data file for the object. The function also closes any index files and/or memo files associated with the data file.

Code fragment from SHOWDATA.CPP

```
   dataFile.close( ) ;
```

Code4::closeAll closes all open data, index, and memo files in the application.

Moving Between Records

CodeBase provides six functions for moving between records in the data file: **Data4::top**, **Data4::bottom**, **Data4::skip**, **Data4::go, Data4::seek** and **Data4::seekNext**. These functions change the current record by loading a new record into the record buffer.

- The **Data4::top** function sets the current record to the first record of the data file. The **Data4::bottom** performs a similar function for the last record.
- The **Data4::skip** function moves a specified number of records above or below the current record. This function must have a current record in order to function correctly.
- The **Data4::go** function loads the record buffer with the record whose record number was provided as a parameter.
- The **Data4::seek** function locates the first record in a sorted order that matches a search key. This function is described in detail in the "Indexing" chapter of this guide.
- The **Data4::seekNext** is similar to **Data4::seek** except that it locates the next record in a sorted order that matches a search key. This function is described in detail in the "Indexing" chapter.

Scanning Through The Records

A common task in database programming is to display a list of records. To accomplish this, the SHOWDATA.CPP program loads each record into the record buffer so its contents can be displayed. This is precisely what the following **for** loop does.

Code fragment from SHOWDATA.CPP

```
for( int rc = dataFile.top( ); rc == r4success ;
                                           rc = dataFile.skip( ))
{
```

Data4::top

In the initialization section of the **for** loop, **Data4::top** is called. This function causes the first record in the data file to be loaded into the record buffer. If the function is successful, it returns a value of **r4success.**

Data4::skip

At the end of each iteration through the **for** loop, **Data4::skip** is called. **Data4::skip** allows you to skip either forwards or backwards through the data file. The numeric argument is used to specify how many records are skipped. If the number is positive, you move towards the end of the data file; if it is negative you move back towards the top. In this example the default value ('1L') is used, which indicates that the next record should be loaded. The **Data4::skip** function also returns a value of **r4success** when it executes successfully.

The **for** loop is terminated when *rc* is no longer equal to **r4success**. This happens when either **Data4::skip** or **Data4::top** encounters an error or the end of the data file.

Accessing Fields

Before any information can be stored to or retrieved from a field, you must first construct a **Field4** or **Field4memo** object. There are two variations of the **Field4/Field4memo** constructor. The first involves knowing the field's position in the data file, and the second requires that you already know the field's name.

PROGRAM CUSTLIST.CPP

This program lists the customer information contained in the DATA1.DBF data file.

```
#include "d4all.hpp"
extern unsigned _stklen = 10000 ;

int main( void )
{
    Code4    codeBase;
    Data4    dataFile( codeBase, "DATA1.DBF" ) ;
    Field4   fNameFld( dataFile, "F_NAME" ),
             lNameFld( dataFile, "L_NAME" ),
             addressFld( d ataFile, "ADDRESS" ),
             ageFld( dataFile, "AGE" ),
             birthDateFld( dataFile, "BIRTH_DATE" ),
             marriedFld( dataFile, "MARRIED"),
             amountFld( dataFile, "AMOUNT" ),
             commentFld( dataFile, "COMMENT" ) ;

    codeBase.exitTest( ) ;
    Date4 birthDate ;
    Str4ten purchased ;
    Str4large name ;
    for( int rc = dataFile.top( ); rc == r4success;  rc = dataFile.skip( ))
    {
        purchased.assignDouble( (double) amountFld, 8, 2 ) ;
        birthDate.assign( birthDateFld.ptr( ) ) ;
        name.assign( fNameFl d ) ;
        name.trim( ) ;
        name.add( " " ) ;
        name.add( lNameFld ) ;

        cout << "------------------------------" << endl ;
        cout << "Name     : " << name.ptr( ) << endl ;
        cout << "Address  : " << addressFld.str( ) << endl ;
        cout << "Age : " << (int) ageFld << " Married : "
             << marriedFld.str( ) << endl ;
        cout << "Birth Date: " << birthDate.format( "MMMMMMMM DD, CCYY" ) << endl ;
        cout << "Comment: " << commentFld.str( ) << endl ;
        cout << "Purchased this year:$" << purchased.ptr( ) << endl ;
    }
    dataFile.close( ) ;
    codeBase.initUndo( ) ;
    return 0 ;
}
```

Referencing By Field Number

Each field in the data file has a unique *field number*. Field numbers range from one to the total number of fields in the data file. This value denotes the field's physical order in the record. The **Field4::Field4** constructor can use a field's field number to reference fields generically.

Iterating through the fields

Iterating through the fields in a record is a common use of this method of field referencing. Before you can do this, **Data4::numFields** is called to determine the number of fields in the data file.

Code fragment from SHOWDATA.CPP

```
   numFields = dataFile.numFields( ) ;
```

A **for** loop is then used to iterate through each field to generically construct a **Field4/Field4memo** object.

```
 for( int j = 1; j <= numFields; j++ )
{
    Field4memo field( dataFile, j ) ;
```

Referencing By Field Name

If you know ahead of time what the names of the fields are, you can construct a **Field4/Field4memo** object using its name. This is the method used in the CUSTLIST.CPP program.

The **Field4/Field4memo** constructor creates an object for the field whose name is specified. This is demonstrated in the following section of code.

Code fragment from CUSTLIST.CPP

```
   Field4        fNameFld( dataFile, "F _NAME" ),
                 lNameFld( dataFile, "L_NAME" ),
                 addressFld( dataFile, "ADDRESS" ),
                 ageFld( dataFile, "AGE" ),
                 birthDateFld( dataFile, "BIRTH_DATE" ),
                 marriedFld( dataFile, "MARRIED" ),
                 amountFld( dataFile, "AMOUNT" ),
                 commentFld( dataFile, "COMMENT" ) ;
```

In this code fragment, a **Field4** object is obtained for each field of the record. These objects may be used at any time while they are in scope.

When the data file is closed, any **Field4** object for the data file becomes invalid.

Retrieving A Field's Contents

The C++ language has many different data types such as **(int), (char), (double),** and **(long)** . As a result, the CodeBase **Str4** class provides a number of operators that retrieve the contents of strings and fields and automatically perform any necessary conversions to the C++ data type.

Retrieving field contents as Strings

If you need to access the field's contents in the form of a null terminated string, you can use **Field4::str** or **Field4memo::str**. They both return **(char *)** pointers to an internal buffer which contains a copy of the field's contents.

Note

Each time a **Str4::str** or a derived class's **str** function is called, the internal CodeBase buffer is overlaid with data from the new object. Consequently, if the object's value (in this case the field's contents) needs to be saved, it is necessary for the application to make a copy of the buffer.

The **Field4::str** function can be used on any type of field except Memo fields. Memo fields may only be accessed using a **Field4memo** object. **Field4memo::str** works on all types of fields including Memo fields. The reason for having two almost identical classes is that Memo fields require special handling. If you know a field is not a Memo field, it is appropriate to use the **Field4** class instead.

To enable the program to work on any type of field, the SHOWDATA.CPP program uses **Field4memo**.

Code fragment from SHOWDATA.CPP

```
Field4memo field( dataFile, j ) ;
cout << field.str( ) ;
```

The CUSTLIST.CPP program uses **Field4::str** on non-Memo fields.

Code fragment from CUSTLIST.CPP

```
cout << "Address  : " << addressFld.str( ) << endl ;
cout << "Age : " << (int) ageFld << " Married : "
     << marriedFld.str( ) << endl ;
```

Character fields are returned by **Field4::str** and **Field4memo::str** as left justified strings padded out with blanks to the full length of the field.

When these functions are used on Date fields, they return the date in the form of an eight character string. If manipulations of the date field are to be made or if they are to be formatted for output, a **Date4** object is usually constructed. Please refer to the "**Date4** Class Functions" chapter for more information.

The contents of Numeric fields are returned as right justified strings padded out to the full length of the field. If the field has any decimal places, the string will contain exactly that many decimal places.

When used on Logical fields, these functions simply return a string containing "Y", "N", "y", "n", "T", "F", "t", or "f" (depending on the value stored on disk in the Logical field).

Retrieving Numerical Data

In addition to returning a field's contents as a string, CodeBase can also convert its contents to numerical data.

Str4::operator double

Str4::operator double returns the contents of the field as a **(double)** value. If **Str4::operator double** is unable to convert the field's contents, a value of **0.0** is returned.

Str4::operator int
Str::operator long

The other two numeric operators, **Str4::operator int** and **Str4::operator long** convert the field's contents to **(int)** or **(long)** values. Any digits to the right of the decimal place (if any are stored in the field) are truncated.

Note

The **Str4** operators operate on any field type, not only dBASE numerics. If, for example, a field is defined as a Character field, but it contains a customer's id number stored as numeric characters, the **Str4** operators can correctly convert it into a numeric value -- regardless of how dBASE interprets it.

PROGRAM
NEWLIST.CPP

The NEWLIST.CPP program creates a new data file if one of the same name does not already exist. It then appends several new records and assigns values to the fields in each record.

```
// NEWLIST.CPP

#include "d4all.hpp"
extern unsigned _stklen = 10000 ; // for all Borland compilers

Code4    codeBase ;
Data4    dataFile ;
Field4   fName, lName, address, age, birthDate, married, amount ;
Field4memo  comment ;

FIELD4INFO  fieldInfo [] =
{
    {"F_NAME",r4str,10,0},
    {"L_NAME",r4str,10,0},
    {"ADDRESS",r4str,15,0},
    {"AGE",r4num,2,0},
    {"BIRTH_DATE",r4date,8,0},
    {"MARRIED",r4log,1,0},
    {"AMOUNT",r4num,7,2},
    {"COMMENT",r4memo,10,0},
    {0,0,0,0},
};

void  OpenDataFile( void )
{
    dataFile.open( codeBase, "DATA1.DBF" ) ;
    if( !dataFile.isValid( ) )
        dataFile.create( codeBase,"DATA1.DBF", fieldInfo ) ;
    fName.init( dataFile, "F_NAME" ) ;
    lName.init( dataFile, "L_NAME" ) ;
    address.init( dataFile,  "ADDRESS" ) ;
    age.init( dataFile,"AGE" ) ;
    birthDate.init( dataFile, "BIRTH_DATE" ) ;
    married.init( dataFile, "MARRIED" ) ;
    amount.init( dataFile, "AMOUNT" ) ;
    comment.init( dataFile, "COMMENT" ) ;
}

void PrintRecords( void )
{
    Date4 bDate ;
    Str4ten purchased ;
    Str4large name ;
    for( int rc = dataFile.top( ); rc == r4success; rc = dataFile.skip( ))
    {
        purchased.assignDouble( (double) amount, 8, 2 ) ;
        bDate.assign( birthDate ) ;
        name.assign( fName ) ;
        name.trim( ) ; name.add( " " ) ;
        name.add( lName ) ;

        cout << "-------------------------------" << endl ;
        cout << "Name     : " << name.ptr( ) << endl ;
        cout << "Address  : " << address.str( ) << endl ;
        cout << "Age : " << (int) age << " Married : "<< married.str( ) << endl ;
    cout << "Birth Date: "<< bDate.format( "MMMMMMMM DD, CCYY" ) << endl ;
    cout << "Comment: " << comment.str( ) << endl ;
        cout << "Purc hased this year:$" << purchased.ptr( ) << endl ;
    }
}

void AddNewRecord(char *fNameStr, char*lNameStr, char *addressStr
             ,char *dateStr, int marriedValue, double amountValue
             ,char *commentStr = NULL )
{
    dataFile.lockAll( ) ;
    dataFile.appendStart( ) ;
    dataFile.blank( ) ;
    fName.assign( fNameStr ) ;
    lName.assign( lNameStr ) ;
    address.assign( addressStr ) ;

    Date4 bDate, today ;
    bDate.assign( dateStr ) ;
    today.today(  ) ;
    // approximate age -- ignore leap year
    long ageValue = ((long) today - (long) bDate) / 365 ;
    age.assignLong( ageValue ) ;
    birthDate.assign( bDate ) ;
    if( marriedValue )
        married.assign( "T" ) ;
    else
        married.assign( "F" ) ;

    amount.assignDouble( amountValue ) ;
    if( commentStr )
        comment.assign( commentStr ) ;

    dataFile.append( ) ;
    dataFile.unlock( ) ;
}
```

```
int main( void )
{
    codeBase.errOpen = 0;
    codeBase.safety = 0;
    codeBase.lockEnforce = 1 ;

    OpenDataFile( ) ;

    PrintRecords( ) ;

    AddNewRecord("Sarah", "Webber", "132-43 St.", "19600223", 1, 147.99,
                                                                      "New Customer");

    AddNewRecord("John", "Albridge", "1232-76 Ave.", "19581012", 0, 98.99 ) ;

    PrintRecords( ) ;

    codeBase.closeAll( ) ;
    codeBase.initUndo( ) ;
    return 0 ;
}
```

FIELD4INFO Structures

The **FIELD4INFO** structure is an integral component of the data file creation process. This structure contains the information which defines a field.

The **FIELD4INFO** structure contains a member for each of the field's four attributes. These members are described below:

- **(char *) name** This is a pointer to a null terminated string containing the name of the field. This name must be unique to the data file and any characters after the first ten are ignored. Valid field names can only contain letters, numbers, and/or underscores. The first character of the name must be a letter.
- **(char) type** This member contains an abbreviation for the field type. CodeBase supports Character, Date, Floating Point, Logical, Memo, Numeric, Binary and General field types. Valid abbreviations for the field types are the character constants 'C', 'D', 'F', 'L', 'M', 'N', 'B' and 'G'. In addition you can also use the equivalent CodeBase constants: **r4str, r4date, r4float, r4log, r4memo, r4num**, **r4bin** and **r4gen**, respectively.
- **(int) len** This integer value determines the length of the field.
- **(int) dec** This member determines the number of decimal places in Numeric or Floating Point fields.

Refer to the **Field4info** class in the CodeBase *Reference Guide* for more detailed information on field attributes.

FIELD4INFO arrays

Before a data file can be created, an array of **FIELD4INFO** structures (or a **Field4info** class object) must be defined. Each element of this array (or field added with **Field4info::add**) describes a single field.

WARNING

Each member of the last element in a **FIELD4INFO** array must be set to null. This indicates that there are no more elements in the array. Failure to provide an element filled with null's will result in errors when the data file is created. The **Field4info** class automatically adds this array element.

An example **FIELD4INFO** array is defined below:

Code fragment from NEWLIST.CPP

```
FIELD4INFO  fieldInfo [] =
{
    {"F_NAME",r4str,10,0},
    {"L_NAME",r4str,10,0},
    {"ADDRESS",r4str,15 ,0},
    {"AGE",r4num,2,0},
    {"BIRTH_DATE",r4date,8,0},
    {"MARRIED",r4log,1,0},
    {"AMOUNT",r4num,7,2},
    {"COMMENT",r4memo,10,0},
    {0,0,0,0},
} ;
```

Alternatively, if the **Field4info** class were used, the same array could be created with the following lines:

```
Field4info fieldInfo( codeBase ) ;
fieldInfo.add( "F_NAME", r4str, 10 ) ;
fieldInfo.add( "L_NAME",r4str, 10 ) ;
fieldInfo.add( "ADDRESS", r4str,15 ) ;
fieldInfo.add( "AGE", r4num, 2 ) ;
ieldInfo.add( "COMMENT", r4memo ) ;
. . .
dataFile.create( codeBase, "DATA1.DBF", fieldInfo.fields( ) ) ;
```

Creating The Data File

The actual creation of the data file is performed by the **Data4::create** function. This function uses the information in a **FIELD4INFO** array as the field specifications for a new data file.

The second parameter of the **Data4::create** function is a pointer to a null terminated string containing a file name. This file name can contain any extension, but if no extension is provided, the default ".DBF" extension is used. Like the **Data4::open** function, the current directory is assumed if no path is provided as part of the file name.

If the data file is successfully created, the information for the data file is stored in the **Code4** object, and the **Data4** object is initialized for the file. If for any reason the data file could not be created, **Data4::isValid** returns a false (non-zero) value.

The optional fourth parameter to **Data4::create** is used for creating production index files. Please refer to the "Indexing" chapter for details.

The **Code4::safety** flag

The **Code4::safety** flag determines whether any existing file is replaced when CodeBase attempts to create a file. If the file exists and the **Code4::safety** flag is set to its default value of true (non-zero), the new file is not created; otherwise when this flag is set to false (zero), the old file is replaced by the newly created file. If the new file is not created because the file already exists, and **Code4::errCreate** is true, an error message is generated.

Opening Or Creating

It is often the case that you want to open a data file if it exists or create it if it does not. The following section of code demonstrates how this can be accomplished:

Code fragment from NEWLIST.CPP

```
Code4 codeBase ;
codeBase.errOpen = 0 ;
codeBase.safety = 0 ;
. . .
dataFile.open( codeBase, "DATA1.DBF" ) ;
if( dataFile.isValid( ) )
    dataFile.create( codeBase,"DATA1.DBF", fieldInfo ) ;
```

The **Code4::errOpen** flag

If the data file does not already exist, **Data4::open** will normally generate an error message when it fails to open the file. This error message can be suppressed by changing the **Code4::errOpen** flag. When this flag is set to its default value of true (non-zero), an error message is generated when CodeBase is unable to open a file. If this flag is set to false (zero), CodeBase does not display this error message. Nevertheless, the object is left uninitialized. An uninitialized **Data4** object may be tested for by calling **Data4::isValid**.

Using the **Code4::errOpen** flag and the **Data4::isValid** function with the **Data4::open** and **Data4::create** provides a simple, guaranteed method of having valid **Data4** objects pointing to open data files.

Adding New Records

There are two methods that can be used to add new records to the data file. They both append a new record to the end of the data file.

Appending A Blank Record

If all you need to do is add a blank record to the bottom of the data file, **Data4::appendBlank** should be used.

```
dataFile.appendBlank( ) ;
```

This function adds a new record to the data file, blanks out all of its fields, and makes the new record the current record.

The Append Sequence

The second method involves three steps. This method is more efficient and should be used when you intend to immediately assign values to the new records fields. Using this method results in less disk writes and therefore improves performance.

The first step is performed by the **Data4::appendStart** function. This function sets the current record number to zero to let CodeBase know that a new record is about to be appended. Additionally, **Data4::appendStart** temporarily disables automatic flushing for the current record, so that if changes need to be aborted, the record is not flushed to disk.

Code fragment from NEWLIST.CPP

```
dataFile.appendStart( ) ;
```

The second step involves assigning values to the fields. After the **Data4::appendStart** function is called, the record buffer contains a copy of the previous current record. If no changes are made to the record buffer, a copy of this record is appended. If you prefer an empty record buffer, a call to **Data4::blank** will clear it.

```
dataFile.appendStart( ) ;
dataFile.blank( ) ;
```

Assigning values to the fields will be discussed in the next section.

Appending the record

The final step is accomplished by **Data4::append**. This function causes a new record to be physically added to the end of the data file and its key values to be added to any open index tags.

Code fragment from NEWLIST.CPP

```
dataFile.append( ) ;
```

Assigning Field Values

The CodeBase **Field** and **Field4memo** class are derived from **Str4** to provide easy access to the fields of a database. Using the same assignment functions documented in the "Using Strings Classes" chapter, the contents of a field may be changed.

In addition to modifying the contents of the record buffer, the **Field4** class overloads **Str4::changed**. The overloaded function then ensures that the data file "record changed" flag reflects any changes made using the **Field4/Field4memo** assignment functions.

Code4::lockEnforce

In multi-user configurations, only one application should edit a record at a time. One way to ensure that only one application can modify a record is to explicitly lock the record before it is changed. To ensure that a record is locked before it is changed, set **Code4::lockEnforce** to true (non-zero). When **Code4::lockEnforce** is true (non-zero) and an attempt is made to modify an unlocked record using a field function or **Data4::blank**, **Data4::changed**, **Data4::delete** or **Data4::recall**, an **e4lock** error is generated and the modification is aborted.

An alternative method of ensuring that only one application modifies a record is to deny all other applications write access to the data file. In this case explicit locking is not required, even when **Code4::lockEnforce** is true, since only no outside applications can write to the data file. Write access can be denied to other applications by setting the **Code4::accessMode** to **OPEN4DENY_WRITE** or **OPEN4DENY_RW** before the data file is opened.

Refer to the "Multi-User Applications" chapter of this guide for more information.

Copying Strings To Fields

The assign function, overloaded for both **Field4** and **Field4memo**, assigns a character string to a field. The **Field4::assign** operates on all field types excluding memo fields. To assign a value to a memo field, use the **Field4memo::assign**.

Copying strings to Character fields

If the length of the assigned string is less than the size of the field, the field's contents are left justified, and the remaining spaces are filled with blanks. If the string is larger than the field, its contents are truncated to fit. When an assignment is done with a memo field, the length is automatically adjusted to accommodate any value.

Code fragment

```
Field4 nameFld( dataFile, "NAME" ) ; // len: 20
Field4memo commentFld( dataFile, "COMMENT") ; // memo

nameFld.assign( "Hubert Horatio Humphry" ) ;
    // nameFld contains "Hubert Horatio Humph" (ie. 20 chars)
commentFld.assign( "Hubert Horatio Humphry III" ) ;
    // comentFld has a length of 26
nameFld.assign( "HHH" ) ;
    // nameFld contains "HHH                 " (ie. 20 chars)
```

Copying strings to Numeric fields

These functions can also store strings into Numeric fields. Generally this is not a good idea since it is very easy to format the information incorrectly. A correctly formatted number is right justified, zero-padded right to the number of decimals in the field, and space-padded left for any unused units.

Note

Str4::assign stores the information in the field exactly as it appears in the string. This can cause problems when used to store non numeric data into Numeric type fields. It is recommended that the **Str4::assignLong** or **Str4::assignDouble** be used to store data in Numeric fields.

Copying strings to Date fields

Any strings that are copied to Date type fields should be exactly eight characters long and should contain a date in standard format (CCYYMMDD). For details please refer to the "**Date4** Class Functions" chapter.

Copying strings to Logical fields

The only strings that should be copied to Logical fields are "T", "F","t", "f","Y", "N", "y" or "n".

Code fragment from NEWLIST.CPP

```
    if( marriedValue )
        married.assign( "T" );
    else
        married.assign( "F" );
```

Storing Numeric Data

Instead of worrying about the formatting of strings containing numerical data, you can let CodeBase perform the formatting for you. **Str4::assignDouble** and **Str4::assignLong** automatically convert and format numerical data.

Assigning **(double)** values to a field

You can assign a **(double)** value to a field using **Str4::assignDouble**. This function can automatically format the numerical value according to the field's attributes. If the last two parameters of **Str4::assignDouble** (*len* and *nDec*) are not specified, the length and the decimals for the field are used. For example, if the field has a length of six and has two decimal places, and the **(double)** value of 34.12345 assigned to it, the field will contain " 34.12" after **Str4::assignDouble** is called. If the **Str4::assignDouble** function is used on a Character field, it assumes the field has no decimals and right justifies its contents.

Code fragment from NEWLIST.CPP

```
 amount.assignDouble( amountValue ) ;
```

Assigning integer values to a field

Str4::assignLong is used to copy and format integer and long values to fields. Right justification is used, and if the field is of type Numeric, any decimal places are filled with zeros.

Code fragment from NEWLIST.CPP

```
age.assignLong( ageValue ) ;
```

Removing Records

CodeBase provides a two level method for removing records. You can delete a record by simply changing the status of its deletion flag to true. The records which are flagged for deletion can then be physically removed by "packing" the data file.

PROGRAM DELETION.CPP

The DELETION.CPP program demonstrates the effects of deleting, recalling, and packing records.

```
#include "d4all.hpp"
extern unsigned _stklen = 10000 ; // for all Borland compilers

FIELD4INFO  fieldInfo[]=
{
    {"DUMMY",'C',10,0},
    {"MEMO",'M',10,0},
    {0,0,0,0},
} ;

void printDeleteStatus(int status,long recNo)
{
    if(status)
        cout << "Record " << recNo << " - DELETED" << endl ;
    else
        cout << "Record " << recNo << " - NOT DELETED" << endl ;
}

void printRecords(Data4 dataFile)
{
    int  rc = 0, status;
    long recNo;

    cout << endl ;

    dataFile.top( ) ;
    while( rc != r4eof )
    {
        recNo = dataFile.recNo( ) ;
        status = dataFile.deleted( ) ;
        printDeleteStatus( status, recNo ) ;
        rc = dataFile.skip( ) ;
    }
}

void main( void )
{
    Code4       codeBase ;
    Data4       dataFile ;
    int count;

    codeBase.safety = 0 ;
    codeBase.errCreate = 0 ;
    codeBase.lockEnforce = 1 ;
    dataFile.create( codeBase, "TUTOR5", fieldInfo, 0 ) ;
    codeBase.exitTest( )  ;

    for(count = 0; count < 5; count ++)
        dataFile.appendBlank( ) ;

    printRecords( dataFile ) ;

    dataFile.lockAll( ) ;
    dataFile.go( 3L ) ;
    dataFile.deleteRec( ) ;
    dataFile.go( 1L ) ;
    dataFile.deleteRec( ) ;
    printRecords( dataFile ) ;

    dataFile.go( 3L ) ;
    dataFile.recall( ) ;
    printRecords( dataFile ) ;
    dataFile.pack( ) ;
    dataFile.memoCompress( ) ;
    printRecords( dataFile ) ;

    codeBase.closeAll( ) ;
    codeBase.initUndo( ) ;
}
```

Explanation: This program creates the data file (or re-creates, if it exists) and then appends five records. The program then displays the deletion status of all the records:

DELETION.CPP output part 1

```
Record #   1 - NOT DELETED
Record #   2 - NOT DELETED
Record #   3 - NOT DELETED
Record #   4 - NOT DELETED
Record #   5 - NOT DELETED
```

Record numbers one and three are then marked for deletion and the deletion status of the records are again displayed.

DELETION.CPP output part 2

```
Record #   1 - DELETED
Record #   2 - NOT DELETED
Record #   3 - DELETED
Record #   4 - DELETED
Record #   5 - NOT DELETED
```

Record number three is recalled (ie. the deletion mark is removed) and the deletion status of the records are displayed.

DELETION.CPP output part 3

```
Record #   1 - DELETED
Record #   2 - NOT DELETED
Record #   3 - NOT DELETED
Record #   4 - DELETED
Record #   5 - NOT DELETED
```

Finally the data file is packed. Displaying the record status shows that record one was removed from the data file. Record two now occupies the space of record one.

DELETION.CPP output part 4

```
Record #   1 - NOT DELETED
Record #   2 - NOT DELETED
Record #   3 - NOT DELETED
```

Note Since the record numbers are contiguous and always start at one, the record number for a specific record may have changed after the data file has been packed.

Determining The Deletion Status

Each record has its own deletion flag. The status of the deletion flag of the current record can be checked using **Data4::deleted**.

Code fragment from DELETION.CPP

```
status = dataFile.deleted( ) ;
```

Data4::deleted returns a true (non-zero) value when the current record has been marked for deletion.

Marking A Record For Deletion

Data4::deleteRec is used to set the current record's deletion status to true. The next time the data file is packed, the record is physically removed from the file and any reference to it in any open index file is also removed.

Code fragment from DELETION.CPP

```
  dataFile.deleteRec( ) ;
```

Note

When a record is marked for deletion, it is not physically removed from the data file. In fact you can still access the record normally. The main importance of the deletion flag is that the marked record is physically removed when the data file is packed. The deletion flag may be used in tag filter expressions. Refer to the "Indexing" chapter for details.

Recalling Records

If the data file has not yet been packed, any deleted records can be recalled back to non-deleted status. This is accomplished by a call to **Data4::recall**.

```
if( dataFile.deleted( ) )
    dataFile.recall( ) ;
```

Packing The Data File

Physically removing records from the data file is called packing. If your data file is quite large, this process can be quite time consuming. That is why records are first marked for deletion and then physically removed. The cumulative time necessary for removing individual records would be substantially greater than removing the many "deleted" records at one time. Packing the data file is performed by **Data4::pack**.

Note Consider opening the file exclusively before packing to keep others from accessing the data file while packing is occurring. Otherwise, the data may appear as corrupted to other users for the duration of the pack.

Compressing The Memo File

When **Data4::pack** removes a record from the data file, it does not remove the corresponding memo entry (assuming that there is at least one Memo field), which results in an unreferenced memo file. This causes the space in the memo file to be wasted. To remove any wasted memo file space, the memo file can be compressed by calling **Data4::memoCompress**.

Unreferenced memo entries cause no difficulties in an application. They simply waste disk space. Again, since compressing a memo file can consume a large amount of time, it may be appropriate to compress a data file sometime other than when the data file is packed.

Note Wasted memo file space is mainly caused by packing records that have memo entries without doing a memo compress. If you are using Clipper compatibility, wasted memo space can also occur when memo entries are increased beyond 504 bytes. In either case, memo compressing reduces disk space usage.

Data File Information

The **Data4** class contains a set of functions which provide information about the data files and their records. **Field4** contains specific information about the field.

PROGRAM DATAINFO.CPP

This program displays information about a data file whose name is provided as a command line argument. It displays the data file's alias, record count, record width, and the attributes of its fields.

```
#include "d4all.hpp"
extern unsigned _stklen = 10000; // for all Borland compilers

int main(int argc,char *argv[])
{
    if(argc != 2)
    {
        printf(" USAGE: DATAINFO <FILENAME.DBF> \n");
        exit(0);
    }
    Code4    codeBase ;
    Data4    dataFile( codeBase, argv[1] ) ;
    codeBase.exitTest( ) ;

    long recCount = dataFile.recCount( ) ;
    int numFields = dataFile.numFields( ) ;
    long recWidth = dataFile.recWidth( ) ;
    const char *alias = dataFile.alias( ) ;

    cout << "\t\t+---------------------------------+" << endl ;
    printf( "\t\tï Data File: %12s         ï\n",argv[1]);
    printf( "\t\tï Alias    : %12s         ï\n",alias);
    printf( "\t\tï                                   ï\n");
    printf( "\t\tï Number of Records: %7ld      ï\n",recCount);
    printf( "\t\tï Length of Record : %7ld      ï\n",recWidth);
    printf( "\t\tï Number of Fields : %7d      ï\n",numFields);
    printf( "\t\tï                                 ï\n");
    printf( "\t\tï Field Information :             ï\n");
    printf( "\t\tï---------------------------------ï\n");
    printf( "\t\tï Name       ï type ï len  ï dec  ï\n");
    printf( "\t\tï-----------+------+------+------ï\n");

    for(int j = 1; j <= dataFile.numFields( ); j ++)
    {
        Field4 field( dataFile, j ) ;
        const char * name = field.name( ) ;
        char type = (char ) field.type( ) ;
        int len = (int) field.len( ) ;
        int dec = field.decimals( ) ;

        printf( "\t\tï %10s ï   %c  ï %4d ï %4d ï\n",name,type,len,dec);
    }
    cout << "\t\t+---------------------------------+" << endl ;

    dataFile.close( ) ;
    codeBase.initUndo( ) ;
    return 0;
}
```

The data file alias

Each data file that you have opened has a character string label called the *alias*. The alias is mainly used for qualifying field names in dBASE expressions or looking up a **Data4** objects with **Code4::data.**

Data4::alias returns a pointer to a string containing the data file's alias. This character pointer is valid as long as the data file is open.

Code fragment for DATAINFO.CPP

```
char *alias = dataFile.alias( ) ;
```

Although you can directly modify the string returned from **Data4::alias**, it is highly recommended that you change the alias by passing the new alias name to **Data4::alias**. This prevents memory corruption that may be caused by writing beyond the end of the string.

Finding the number of records

An important data file statistic is the record count. You can obtain this value using **Data4::recCount**. This tells you how many records are in the data file. This function does not take into account the deleted status of any records, nor any filter condition of a open tag.

Code fragment for DATAINFO.CPP

```
   long recCount = dataFile.recCount( ) ;
```

Determining the record width

Another useful function is **Data4::recWidth**. This function returns the length of the record buffer. This value is the total length of all the fields plus one byte for the deletion flag. This function may be used to save copies of the record buffer, or for use with the **Str4len** constructor where a length is required.

Code fragment for DATAINFO.CPP

```
long  recWidth = dataFile.recWidth( ) ;
```

Field Information

If you have a constructed **Field4** object, you can retrieve the field attributes for the referenced field.

Field name

Field4::name returns a pointer to the field's name.   This pointer is valid as long as the data file is open.

Code fragment for DATAINFO.CPP

```
char *name = field.name( ) ;
```

WARNING

Do not modify the string returned by **Field4::name.** Doing so corrupts CodeBase.

Field type

Field4::type returns the field type:

```
switch( field.type( ) )
{
    case 'M':
        cout << "Memo" ; break ;
    case 'C':
        cout << "Character" ; break ;
    case 'N':
    case 'F':
        cout << "Numeric" ; break ;
    case 'D':
        cout << "Date" ; break ;
    case 'L':
        cout << "Logical" ; break ;
}
```

Field length and decimals

The length of the field and the number of decimals are returned by **Field4::len** (**Field4memo::len**) and **Field4::decimals** respectively.

Advanced Topics

This section explains the use multiple **Code4** objects in an application, and copying data file structures.

Multiple Code4 Objects

Normally, you have one **Code4** object in your application. However, sometimes your application needs to access data files only in a few places for a short time. In this case, it can be appropriate to have more than one **Code4** object.

Uninitializing the Code4 object

If you have a single **Code4** object that is constructed and initialized at the beginning of the application, you can tie up memory that you are not using. To avoid this, you can initialize the **Code4** object before you start using it and uninitialize it after. This is done through calls to **Code4::init** and **Code4::initUndo**.

Code4::init initializes CodeBase internal memory and sets all of the **Code4** member variables to their default values.

Code4::initUndo closes all data, memo, and index files. It returns any allocated memory back to the CodeBase memory management pool or to the operating system.

```
void Function( void )
{
    Code4   codeBase;
    Data4   dataFile( codeBase, "DATAFILE" ) ;

    ...  perform CodeBase operations ...

    codeBase.initUndo( ) ;
}

void OtherFunction( void )
{
    Code4   codeBase ;

    ...  Do some other CodeBase operations ...

    codeBase.initUndo( ) ;
}

void main( void )
{

    Function( );
    OtherFunction( );

    ... etc ...
}
```

Copying Data File Structures

You may occasionally require a new data file that has an identical structure as an existing data file.

To satisfy this need CodeBase provides class for dynamically building the **FIELD4INFO** structure. This class is **Field4info**. A **Field4info** constructor can be used to copy the structure of a data file. Once constructed, the original data file may be closed, if necessary, and the new data file may be created with **Data4::create.**

The **FIELD4INFO** array stored in the **Field4info** class is allocated dynamically and should therefore be deallocated after you are finished using it. **Field4info::free** may be called to free up the memory, otherwise the **Field4info** destructor may be called implicitly.

PROGRAM COPYDATA.CPP

The COPYDATA.CPP program takes two file names as arguments. It uses the structure from the data file specified by the first argument to create an empty data file with the name specified by the second argument.

```
#include "d4all.hpp"
extern unsigned _stklen = 10000 ; // for all Borland compilers

void main(int argc,char *argv[])
{
    if(argc != 3)
    {
        cout << "USAGE: COPYDATA <FROM FILE>  <TO FILE>" << endl ;
        exit(1);
    }
    Code4      codeBase;
    Data4      dataFile( codeBase, argv[1] ), dataCopy ;
    codeBase.exitTest( ) ;

    Field4info fields( dataFile ) ;
    codeBase.safety = 0;
    dataCopy.create( codeBase,argv[2], fields.fields( )) ;
    fields.free( ) ;

    codeBase.closeAll( ) ;
    codeBase.initUndo( ) ;
}
```

Field4info::fields returns a copy of the **FIELD4INFO** structure for the data file passed its constructor. Once the **Field4info** is constructed the original data file may be closed, since the **Field4info** constructor makes a copy of the file's structure.

In COPYDATA.CPP, **Field4info::free** is called to deallocate the structure. If it hadn't been called, the **Field4info** destructor would have freed up all of the memory associated with the copy of the data file structure.

5 Indexing

The purpose of a database is to organize information in a useful manner. So far you have seen how the information is divided into fields and records. Now it is time to order the records in purposeful ways. One approach is to physically sort records in your data files. Unfortunately, maintaining data file sorts involves continually shuffling large amounts of data. In addition, it is only possible to maintain a single sort order using this method.

A preferable method is to leave the records in the data file in their original order and store the sorted orderings in a separate file called an index file. When you create an index file, you are effectively sorting the data file. Index files are efficiently maintained and you can have an unlimited number of sorted orderings continually available.

Indexes & Tags

Each index file can contain one or more sorted orderings. These sorted orderings are identified by a *tag*. That is, each index file tag corresponds to a single sorted ordering. CodeBase supports three types of index files. Their attributes and differences are described below:

File Format	Compatibility	Number of Tags	Production Indexes	Group Files	Index Filtering	Descending Ordering
.MDX	dBASE IV	1 - 47	yes	no	yes	yes
.CDX	FoxPro 2.x FoxPro 3.0	1 - 47	yes	no	yes	yes
.NTX	Clipper 87 Clipper 5	1	Only with group files	yes	no	Special Case

Figure 5.1 Index File Formats

The difference between indexes and tags are illustrated in Figure 5.2.

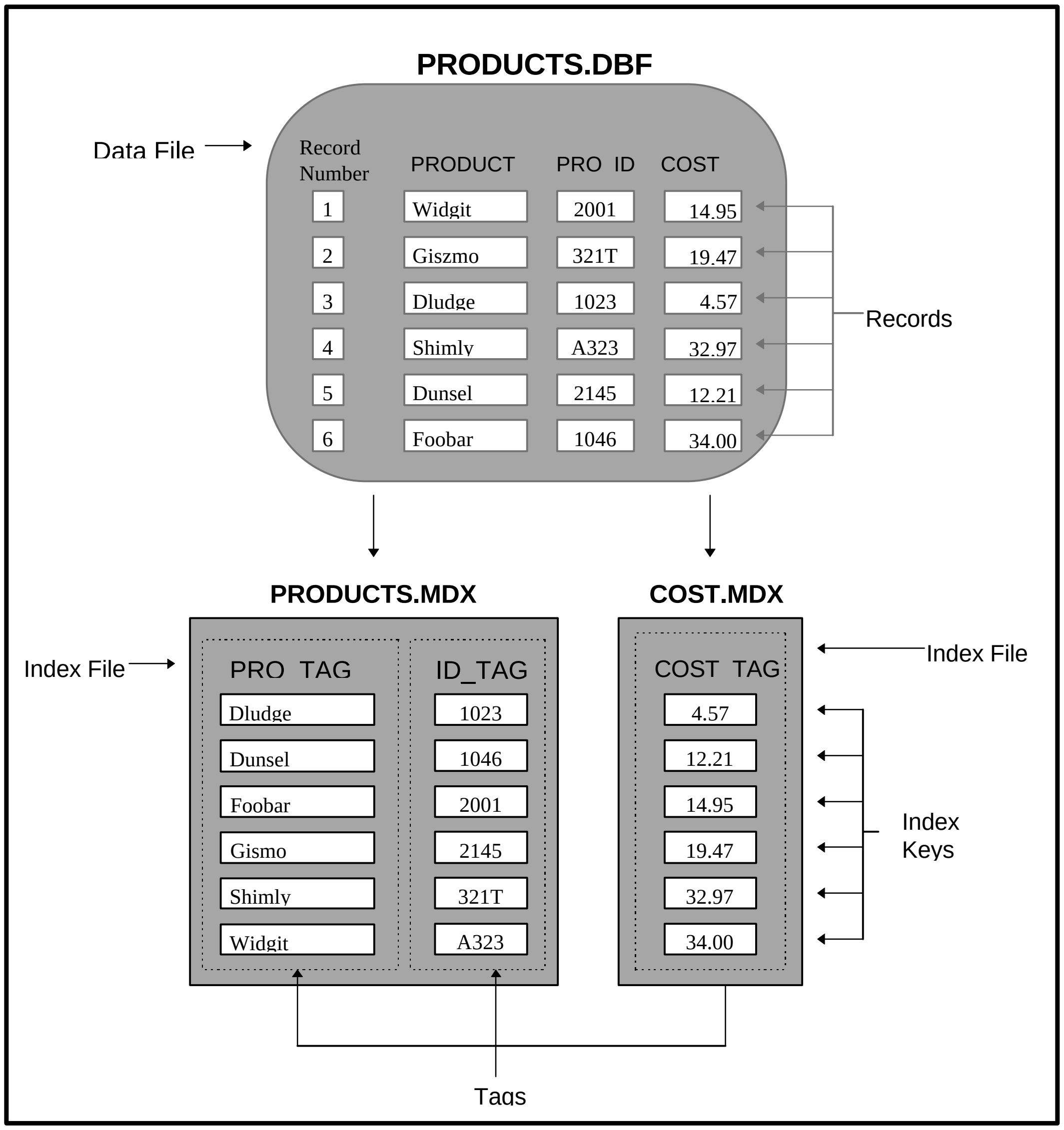


Figure 5.2 Index Files

Index Expressions

An *index expression* is a *dBASE expression* that is used to determine the *index key* for each record. An index expression must evaluate to a value of one of the following types: Numeric, Character, or Date types. When you are using **.CDX** (FoxPro) indexes, you can also index on Logical expressions. Refer to "Appendix C: dBASE Expressions" in the CodeBase *Reference Guide* for details on dBASE expressions.

The most commonly used index expression is a field name. For example, the index expression for the PRO_TAG tag in Figure 5.2 is "PRODUCT". When this expression is evaluated for record number 6, the value of the field PRODUCT is returned; for record number 6, this value is "Foobar   ". To put it simply, tag PRO_TAG is ordered by the contents in field PRODUCT.

Other common index expressions involve generating tags based on two or more fields. This is known as a *compound index key*. For example, if we assume that field PRO_ID is also a character field, we can base a tag ordering with the following index expression: "PRODUCT + PRO_ID". This produces an index key consisting of the PRODUCT field concatenated with the PRO_ID field. For record number 6, the resulting index key is "Foobar   1046".

In addition, most *dBASE functions* are also permitted. See "Appendix C: dBASE Expressions" in the CodeBase *Reference Guide* for a list of dBASE functions supported. If we wanted to base a tag on the PRODUCT field, but remain case insensitive, we can use the dBASE function UPPER() to convert the index key to upper case. In this case the index expression would be "UPPER(PRODUCT)".

Creating An Index File

Creating an index file is similar to creating a data file; in fact you can even create both at the same time. There are two types of index files that you can create: *production indexes* and *non-production indexes*.

PROGRAM NEWLIST2.CPP

The following is the program listing for NEWLIST2.CPP. This modified version of the NEWLIST.CPP program, creates an index file along with the data file.

```
/* NEWLIST2.CPP */
#include "D4ALL.HPP"

Code4    codeBase ;
Data4    dataFile ;
Field4   fName, lName, address, age, birthDate, married, amount;

Field4memo comment ;
Tag4      nameTag, ageTag, amountTag ;

FIELD4INFO  fieldInfo [] =
{
    {"F_NAME",r4str,10,0},
    {"L_NAME",r4str,10,0},
    {"ADDRESS",r4str,15,0},
    {"AGE",r4num,2,0},
    {"BIRTH_DATE",r4date,8,0},
    {"MARRIED",r4log,1,0},
    {"AMOUNT",r4num,7,2},
    {"COMMENT",r4memo,10,0},
    {0,0,0,0},
};

TAG4INFO  tagInfo[] =
{
    {"NAME_TAG","F_NAME + L_NAME",0,0,0},
    {"AGE_TAG","AGE",0,0,0},
    {"AMNT_TAG","AMOUNT",0,0,0},
    {0,0,0,0,0},
};

void  CreateDataFile( void )
{
    dataFile.create( codeBase,"DATA1.DBF", fieldInfo, tagInfo ) ;

    fName.init( dataFile, "F_NAME");
    lName.init( dataFile, "L_NAME");
    address.init( dataFile, "ADDRESS");
    age.init( dataFile, "AGE");
    birthDate.init( dataFile, "BIRTH_DATE");
    married.init( dataFile, "MARRIED");
    amount.init( dataFile, "AMOUNT");
    comment.init( dataFile, "COMMENT");

    nameTag.init( dataFile, "NAME_TAG");
    ageTag.init( dataFile, "AGE_TAG");
    amountTag.init( dataFile, "AMNT_TAG");
}

void PrintRecords( void )
{
    Date4 bDate ;
    Str4ten purchased ;
    Str4large name ;

    for( int rc = dataFile.top( ); rc == r4success; rc = dataFile.skip( ))
    {
        purchased.assignDouble( (double) amount, 8, 2 ) ;
        bDate.assign( birthDate ) ;
        name.assign( fName ) ;
        name.trim( ) ;
        name.add( " " ) ;
        name.add( lName ) ;
        cout << "-------------------------------" << endl ;
        cout << "Name     : "  << name.ptr( ) << endl ;
        cout << "Address  : " << address.str( ) << endl ;
        cout << "Age : " << (int) age << " Married : "
             << married.str( ) << endl ;
        cout << "Birth Date: "
             << bDate.format( "MMMMMMMM DD, CCYY" ) << endl ;
        cout << "Comment: " << comment.str( ) << endl ;
        cout << "Purchased this year:$" << purchased.ptr( ) << endl ;
    }
}

void AddNewRecord(char *fNameStr, char*lNameStr, char *addressStr,
                        char *dateStr, i nt marriedValue, double amountValue,
                        char *commentStr = NULL )
{
    dataFile.lockAll( ) ;
    dataFile.appendStart( ) ;
    dataFile.blank( ) ;

    fName.assign( fNameStr ) ;
    lName.assign( lNameStr ) ;
    address.assign( addressStr ) ;

    Date4 bDate, today ;
    bDate.assign( dateStr ) ;
    today.today( ) ;
    // approximate age -- ignore leap year
    long ageValue = ((long) today - (long) bDate) / 365 ;
    age.assignLong( ageValue ) ;
    birthDate.assign( bDate ) ;

    if( marriedValue )
        married.assign( "T" ) ;
    else
        married.assign( "F" ) ;

    amount.assignDouble( amountValue ) ;
    if( commentStr )
        comment.assign( commentStr ) ;

    dataFile.append( ) ;
    dataFile.unlock( ) ;
}

int main( void )
{
    codeBase.safety = 0 ;
    codeBase.lockEnforce = 1 ;

    CreateDataFile( ) ;

    AddNewRecord( "Sarah", "Webber", "132-43 St.", "19600212", 1, 147.99,
                                                                      "New Customer");
    AddNewRecord("John", "Albridge", "1232-76 A  ve.", "19581023", 0, 98.99 ) ;

    PrintRecords( ) ;

    dataFile.select( nameTag ) ;
    PrintRecords( ) ;

    dataFile.select( ageTag ) ;
    PrintRecords( ) ;
    dataFile.select( amountTag ) ;
    PrintRecords( ) ;

    codeBase.closeAll( ) ;
    codeBase.initUndo( ) ;
    return ;
}
```

Production Indexes

A production index is an index that is opened automatically when its associated data file is opened. A data file can have only one production index and this index has the same name as the data file, but with a different extension.

Non-Production Indexes

All other index files are non-production indexes. A data file can have an unlimited number of non-production indexes. To use these indexes, they must explicitly be opened after the data file has been opened.

The TAG4INFO Structure

Just as the attributes of a field in the data file are defined by a **FIELD4INFO** structure, the attributes of a tag in an index file are specified by a **TAG4INFO** structure. This structure contains the following five members:

The **TAG4INFO** members

The first two members must be defined for every **TAG4INFO** structure. The last three members are used to specify optional properties of the tag which will be discussed later in this chapter.

- **(char *) name** This is a pointer to a character array containing the name of the tag. The name may be composed of letters, numbers and underscores. Only letters and underscores are permitted as the first character of the name. This member cannot be null except to indicate that there are no more tags.

  When using FoxPro or **.MDX** formats, the name must be unique to the data file and have a length of ten characters or less excluding the extension. If you are using the **.NTX** index format, then this name can include a tag name with a path. In this case, the tag name within the path is limited to eight characters or less, excluding the extension.

- **(char *) expression** This is a **(char)** pointer to the tag's index expression. This expression determines the sorting order of the tag. (see the "Index Expressions" section of this chapter for details) This member cannot be null.

- **(char *)filter** This is a pointer to the tag's filter expression.
- **(int) unique** This integer determines how duplicate keys are treated. See the "Unique Keys" section of this chapter for more details.
- **(int) descending** This integer determines if the index should be generated in descending order. Set this member to **0** to specify ascending order or **r4descending** if descending order is desired.

Before the index file can be created, an array of **TAG4INFO** structures must be defined. Each element of this array denotes a single tag.

The **Tag4info** class

The **TAG4INFO** structure may be created dynamically by using the **Tag4info** class. This class has several constructors which can be used to copy the tag structures of existing files. In addition **Tag4info** has functions to dynamically add and delete tag entries in this array.

Note

Each member of the last element in a **TAG4INFO** array must be set to null. This indicates that there are no more elements in the array. Failure to provide an element filled with null's will result in errors when the index file is created. The **Tag4info** class automatically null terminates the array.

An example **TAG4INFO** array is defined below.

Code fragment from NEWLIST2.CPP

```
TAG4INFO  tagInfo[ ] =
{
    "NAME_TAG","F_NAME + L_NAME", 0, 0, 0},
    "AGE_TAG","AGE", 0, 0, 0},
    "AMNT_TAG","AMOUNT",0 ,0, 0},
    {0,0,0,0,0},
};
```

This same **TAG4INFO** array may be built dynamically with the **Tag4info** class with the following code.

```
Tag4info  tagInfo ;

tagInfo.add( "NAME_TAG", "F_NAME + L_NAME" ) ;
tagInfo.add( "AGE_TAG", "AGE" ) ;
tagInfo.add( "AMNT_TAG", "AMOUNT" ) ;
```

Using Data4::create

The most common method for creating an index file is to create it when the data file is created. In this case the index will be a

To Create Index Files

production index. If you recall, the **Data4::create** accepts four parameters, only three of which were used in the last chapter. The fourth parameter is a pointer to a **TAG4INFO** array which is used for creating a production index file. This is illustrated in the following code segment:

Code fragment from NEWLIST2.CPP

```
  dataFile.create( codeBase, "DATA1.DBF", fieldInfo, tagInfo) ;
```

Using Index4::create To Create Index Files

The other method for creating index files involves the use of the **Index4::create** function. In addition to a **TAG4INFO** array, this function also accepts a string containing a file name. An index file is created with this file name. If no extension is specified, an appropriate extension for the file compatibility used (**.MDX**, **.CDX**, or **.NTX**) is provided. For example, the program NEWLIST2.CPP could be modified to use **Index4::create** instead.

```
dataFile.create( codeBase, "DATA1.DBF", fieldInfo ) ;
Index4  index  ;
index.create(dataFile, "DATAINTX", tagInfo) ;
```

Index4::create initializes the **Index4** object with the newly created index file. This object may be used to access many lower level index functions.

Creating Production Indexes with **Index4::create**

Index4::create(Data4, char *, TAG4INFO *) does not produce production indexes, even if the file name that you provide is the same as the data file's. **Index4::create** can be used to create production indexes in the following manner. The data file must be opened exclusively before **Index4::create** is called. This can be achieved by setting the **Code4::accessMode** to **OPEN4DENY_RW** before the data file is opened. After the data file is opened or created exclusively, then **Index4::create(Data4, TAG4INFO *)** is used to create a production index. This is illustrated as follows:

```
codeBase.accessMode = OPEN4DENY_RW;
dataFile.create( codeBase, "DATA1.DBF", fieldInfo ) ;
Index4  index  ;
index.create(dataFile, tagInfo) ;
```

Maintaining Index Files

CodeBase automatically updates all open index files for the data file. When a record is added, modified, or deleted, the appropriate tag entries for all of the open index tags are modified automatically. In general application programming, there is no need to manually add, delete, or modify the tag entries. CodeBase handles all of the key manipulation in the background, letting you concentrate on application programming.

Code fragment

```
Code4 codeBase ;
Data4 dataFile( codeBase, "INFO" ) ;
Field4 nameField( dataFile, "NAME" ) ;
dataFile.top( ) ;
nameField.assign( "ROBERT WILKHAM") ;
dataFile.skip( ) ;
// the changes are flushed to disk.  If "NAME" is a
// key field, the index tag entry is automatically
// updated.  A seek for "ROBERT WILKHAM" would succeed
```

Opening Index Files

Opening index files can be accomplished by one of two functions. If the data file has a production index, **Data4::open** or **Data4::Data4** automatically open it. Other index files may be opened using function **Index4::open** or **Index4::Index4**:

```
Code4   codeBase ;
Data4   dataFile( codeBase, "DATAFILE" ) ;
Index4 index( dataFile, "INDXFILE" ) ;
```

Disabling the automatic opening of production indexes

There may be occasions when you prefer to leave the production index file closed. You can disable the automatic opening of production index files by setting flag **Code4::autoOpen** to false (zero). The production index can then be opened like any other index file:

```
Code4   codeBase ;
codeBase.autoOpen = 0 ;
Data4   dataFile( codeBase, "DATAFILE" ) ;
Index4  index( dataFile, "INDXFILE" ) ;
Index4  index2( dataFile, "DATAFILE" ) ;
```

Referencing Tags

Before a tag is used, you must first construct a **Tag4** object for the data file. Once constructed, the object may be used to select its sort order or to directly manipulate the tag.

PROGRAM SHOWDAT2.CPP

The program SHOWDAT2.CPP is a modified version of SHOWDATA.CPP. This version displays the records of the data file in natural order and according to any tags in its production index.

```
/* SHOWDAT2.CPP */
#include "d4all.hpp"

Code4    codeBase ;
Data4    dataFile ;
Tag4     tag ;

void printRecords( void )
{
    for(int rc = dataFile.top( ); rc == r4success; rc = dataFile.skip( ))
    {
        for(int j = 1;j <= dataFile.numFields( ); j ++)
        {
            Field4memo fie ld( dataFile, j ) ;
            cout << field.str( ) ;
        }
        cout << endl ;
    }
    cout << endl ;
}

void main( int argc, char *argv[] )
{
    if(argc != 2) {  exit(0); }

    dataFile.open( codeBase, argv[1] ) ;
    codeBase.exitTest( ) ;
    cout << "Data File " << argv[1] << " in Natural Order" << endl ;

    printRecords( ) ;
    for( tag.initFirst( dataFile ); tag.isValid( ); tag.initNext( ))
    {
        cout << "Press ENTER to continue:" ;
        getchar( );
        cout << endl << "Data File " << argv[1] << " sorted by Tag "
         << tag.alias( ) << endl ;
        dataFile.select( tag ) ;
        printRecords( ) ;
    }

    dataFile.close( ) ;
    codeBase.initUndo( ) ;
}
```

Referencing By Name

If you know the tag's name, you can construct a **Tag4** object explicitly. **Tag4::Tag4** and **Tag4::init** look through all the open index files of the specified data file for a tag matching the name. This method is used by program 'NEWLIST2.CPP'.

Code Fragment from NEWLIST2.CPP

```
nameTag.init( dataFile, "NAME_TAG" ) ;
ageTag.init( dataFile, "AGE_TAG" ) ;
amountTag.init( dataFile, "AMNT_TAG" ) ;
```

Iterating Through The Tags

There are several **Tag4** member functions which may be used to iterate through the list of tags for the data file. They are: **Tag4::initFirst, Tag4::initLast, Tag4::initNext, Tag4::initPrev**.

An object may be initialized to the first tag or the last tag in the list of tags for the data file by using **Tag4::initFirst** and **Tag4::initLast** respectively. These functions should be used when iterations through the list of tags occurs from the beginning or end of the list.

Tag4::initNext initializes the tag to the next tag after the object's tag. If an iteration is attempted outside the bounds of the tag list (ie. **Tag4::initPrev** on the first tag or **Tag4::initNext** on the last tag), the tag is de-initialized and **Tag4::isValid** returns a false (zero) value.

Code fragment from SHOWDAT2.CPP

```
for( tag.initFirst(dataFile); tag.isValid( ); tag.initNext( ))
{
    . . .
}
```

Selecting Tags

Initially a data file does not have a selected tag and is therefore in *natural order*.   Natural order is the order in which the records were added to the data file. When you want to use a particular sort ordering, select a tag by calling **Data4::select**.

Code fragment from NEWLIST2.CPP

```
dataFile.select( nameTag ) ;
PrintRecords( ) ;

dataFile.select( ageTag ) ;
PrintRecords( ) ;

dataFile.select( amountTag ) ;
PrintRecords( ) ;
```

Selecting Natural Order

If you want to return to natural order simply pass an uninitialized **Tag4** object to **Data4::select**.

```
...

dataFile.select( nameTag ) ;

... do something with the names ...


Tag4 natural ; // uninitialized.
dataFile.select( natural ) ;  // return to natural order
```

Obtaining the currently selected tag

Tag4::initSelected initializes the tag object with the currently selected tag. If no tag is currently selected, **Tag4::initSelected** de-initializes the tag. **Tag4::isValid** may be used to determine if a tag was selected.

The Effects Of Selecting a Tag

Once a tag is selected, the behaviour of **Data4::top, Data4::bottom**, and **Data4::skip** changes.   Functions **Data4::top** and **Data4::bottom** then set the current record to the first and last data file entries respectively, using the tag's sort ordering. Similarly, **Data4::skip** skips using the tag ordering instead of the natural ordering.

Tag Filters

A tag filter is used to obtain a subset of the available data file records. The subset is based on true/false conditions calculated from a dBASE expression. Only records that pass through the filter have entries in the tag. A tag filter is created when the tag is created and cannot later be changed without destroying and recreating the tag.

PROGRAM SHOWLST2.CPP

In program 'SHOWLST2.CPP', when the index file is created, several filters are present.

```
#include "d4all.hpp"

#ifdef __TURBOC__
    extern unsigned _stklen = 10000;
#endif

Code4    codebase ;
Data4    dataFile ;
Field4   fName, lName, address, age, birthDate, married, amount ;
Field4memo   comment ;
Tag4      nameTag, ageTag, amountTag, addressTag, birthdateTag ;

TAG4INFO  tagInfo[ ] =
{
    {"NAME", "L_NAME+F_NAME", ".NOT. DELETED()", 0, 0},
    {"ADDRESS", "ADDRESS", 0, 0, 0},
    {"AGE_TAG", "AGE", "AGE >= 18", 0, 0},
    {"DATE_TAG", "BIRTH_DATE",  0, 0, 0},
    {"AMNT_TAG", "AMOUNT", 0, 0, 0},
    {0, 0, 0, 0, 0},
};

void  OpenDataFile( void )
{
    codeBase.accessMode = OPEN4DENY_RW ;
    codeBase.autoOpen = 0;
    codeBase.safety = 0;

    dataFile.open( codeBase, "DATA1.DBF" ) ;
    Index4 index ;
    index.create( dataFile, NULL, tagInfo ) ;

    fName.init( dataFile, "F_NAME" ) ;
    lName.init( dataFile, "L_NAME" ) ;
    address.init( dataFile, "ADDRESS" ) ;
    age.init( dataFile, "AGE" ) ;
    birthDate.init( dataFile, "BIRTH_ DATE" ) ;
    married.init( dataFile, "MARRIED" ) ;
    amount.init( dataFile, "AMOUNT" ) ;
    comment.init( dataFile, "COMMENT" ) ;

    nameTag.init( dataFile, "NAME" ) ;
    addressTag.init( dataFile, "ADDRESS" ) ;
    ageTag.init( dataFile, "AGE_TAG" ) ;
    birthdateTag.init( dataFile, "DATE_TAG" ) ;
    amountTag.init( dataFile, "AMNT_TAG" ) ;
}

void PrintRecords( void )
{
    Date4 bDate ;
    Str4ten purchased ;
    Str4large name ;

    for( int rc = dataFile.top( ); rc == r4success; rc = dataFile.skip( ))
    {
        purchased.assignDouble( (double) amount, 8, 2 ) ;
        bDate.assign( birthDate ) ;
        name.assign( fName ) ;
        name.trim( ) ;
        name.add( " " ) ;
        name.add( lName ) ;
        cout << "\t\t-------------------------------" << endl ;
        cout << "\t\tName     : " << name.ptr( ) << endl ;
        cout << "\t\tAddress  : " << address.str( ) << endl ;
        cout << "\t\tAge : " << (int) age  << " Married : "
                                  << married.str( ) << endl ;
        cout << "\t\tBirth Date: "
                                  << bDate.format( "MMMMMMMM DD, CCYY" ) << endl ;
        cout << "\t\tComment: " << comment.str( ) << endl ;
        cout << "\t\tPurchased this year:$" << purchased.ptr( ) << endl ;
    }
}

void main( void )
{
    OpenDataFile( ) ;
    dataFile.select( nameTag ) ;

    cout << "Order by Name" << endl ;
    PrintRecords( ) ;
    getchar( ) ;

    dataFile.select( ageTag ) ;
    cout << "Order by Age" << endl ;
    PrintRecords( ) ;
    getchar( ) ;

    dataFile.select( amountTag ) ;
    cout << "Order by Amount" << endl ;
    PrintRecords( ) ;
    getchar( ) ;

    codeBase.closeAll( ) ;
    codeBase.initUndo( ) ;
}
```

Filter Expressions

A *filter expression* is a dBASE expression that returns a Logical result and is used as a tag filter. The expression is evaluated for each record as its tag entry is updated. If the filter expression evaluates to true, an entry for that record is included in the tag. If it evaluates to false, that record's tag entry is omitted from the tag.

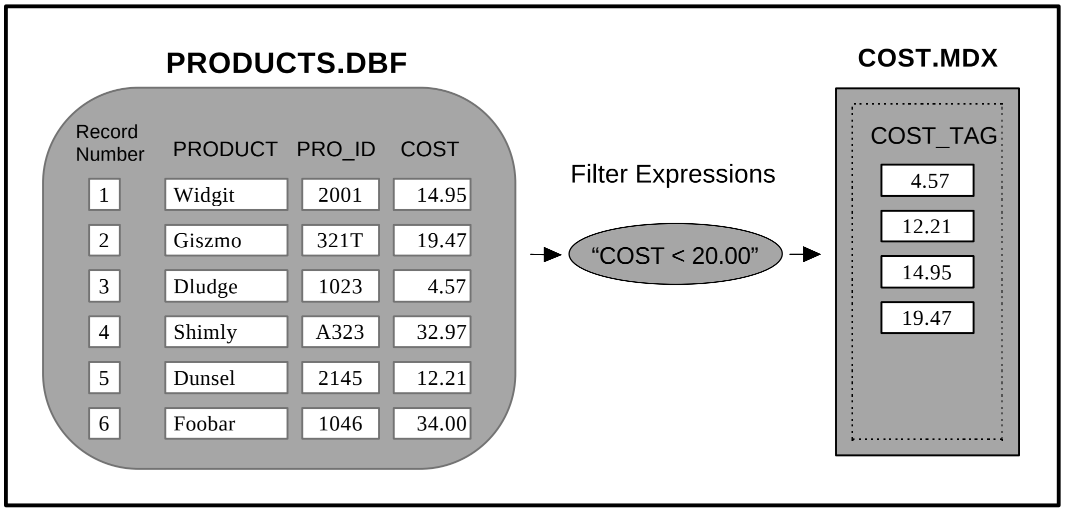


Figure 5.3 Filter Expressions

Creating A Tag Filter

The tag filter is specified, when the index is initially created, through the **TAG4INFO.filter** structure member**.** If this member is null, then no records are filtered. The following code segment shows several tags with filters:

Code Fragment from SHOWDAT2.CPP

```
TAG4INFO  tagInfo[] =
{
    {"NAME","L_NAME+F_NAME", ".NOT. DELETED()",0,0},
    {"ADDRESS","ADDRESS",0,0,0},
    {"AGE_TAG","AGE", "AGE >= 18",0,0},
    {"DATE_TAG","BIRTH_DATE",0,0,0},
    {"AMNT_TAG","AMOUNT",0,0,0},
    {0,0,0,0,0},
} ;
```

The first tag has a filter expression of ".NOT. DELETED()". This filters out any records that have been marked for deletion. The third tag's filter expression is "AGE >= 18". This excludes any record which has an AGE field value of less than eighteen.

Unique Keys

The fourth member of the **TAG4INFO** structure, **(int) unique**, determines how duplicate keys are handled. This member can have four possible values: **0**, **r4uniqueContinue, e4unique** and **r4unique.**

It is often desirable to ensure that no two index keys are the same. To meet this requirement, you can specify that a tag must only contain unique keys. As a result, when a new key is to be added to the tag, a search is first made to see if that key already exists in the tag. If it does exist, the new entry is not added. CodeBase provides three methods of handling the addition of non-unique keys: **r4uniqueContinue, e4unique** and **r4unique.**

The descriptions of the values for the unique member are as follows:

- **0** Duplicate keys are allowed.
- **r4uniqueContinue** Any duplicate keys are discarded. However, the record continues to be added to the data file. In this case, there may not be a tag entry for a particular record.
- **e4unique** An error message is generated if a duplicate key is encountered.
- **r4unique** The data file record is not added or changed. Regardless, no error message is generated. Instead a value of **r4unique** is returned.

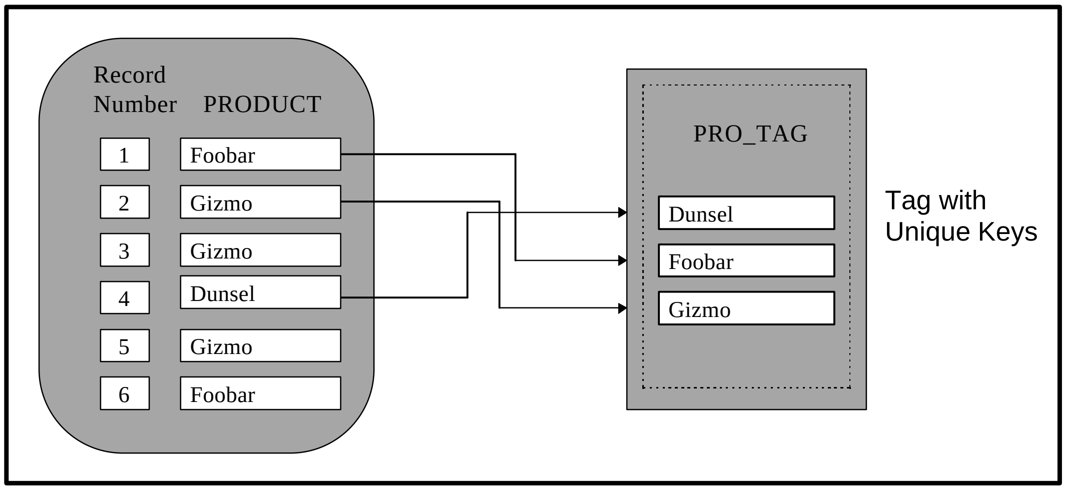


Figure 5.4 Unique Keys

Here is an example **TAG4INFO** array that uses unique keys:

```
TAG4INFO  tagInfo[ ] =
{
    {"NAME","L_NAME + F_NAME",0, r4unique,0},
    {"AGE_TAG","AGE","AGE >= 18",0,0},
    {"AMNT_TAG","AMOUNT",0, r4uniqueContinue,0},
    {0,0,0,0,0},
};
```

WARNING

Although a specific unique code for a tag is specified when the tag is created, only a true/false flag that indicates whether a tag is unique or non-unique is actually stored on disk. No information on how to respond to a duplicate key for unique key tags is saved.
As a result, when a tag is re-opened, the value contained in **Code4::errDefaultUnique** is used to set the tag's unique code. If the tag was designed to be **r4unique** or **e4unique**, then the default value of **r4uniqueContinue** must be changed before any records are modified or appended.
This change can be accomplished in two ways. The first involves setting the **Code4::errDefaultUnique** to another value, in which case all indexes opened subsequently will have this value for their unique code. This setting has no effect on non-unique tags.
The second method is to set the unique code for one or more tags individually after the index file has been opened. Pass the appropriate unique code to the function **Tag4::unique**. This function only has an effect on unique tags.

Note

void Tag4::unique(int uniqueCode) can only be used to change the setting of a unique tag. Setting a unique tag to a non-unique tag or setting a non-unique tag to a unique tag will generate a CodeBase error.

Seeking

One of the most useful features of a database is the ability to find a record by providing a *search key*. This process is referred to as *seeking*. When a seek is performed, the search key, which is usually a string, is compared against the index keys in the selected tag. When a match occurs, the corresponding record is loaded in the record buffer.

PROGRAM SEEKER.CPP

The SEEKER.CPP program demonstrates seeks on the various types of tags. When it performs the seeks, it displays the record that was found and the return value that was returned from the seek.

```
#include "d4all.hpp"
extern unsigned _stklen = 10000 ;

Code4    codeBase ;
Data4    dataFile ;
Field4   fName, lName, address, age, birthDate, married, amount ;
Field4memo comment ;
Tag4     nameTag, ageTag, amountTag, addressTag, birthdateTag;

void  OpenDataFile( void )
{
    dataFile.op en( codeBase, "DATA1.DBF" ) ;
    fName.init( dataFile, "F_NAME" ) ;
    lName.init( dataFile, "L_NAME" ) ;
    address.init( dataFile, "ADDRESS" ) ;
    age.init( dataFile, "AGE" ) ;
    birthDate.init( dataFile, "BIRTH_DATE" ) ;
    married.init( dataFile, "MARRIED" ) ;
    amount.init( dataFile, "AMOUNT" ) ;
    comment.init( dataFile, "COMMENT" ) ;

    nameTag.init( dataFile, "NAME" ) ;
    addressTag.init( dataFile, "ADDRESS" ) ;
    ageTag.init( dataFile, "AGE_TAG" ) ;
    birthdateTag. init( dataFile, "DATE_TAG" ) ;
    amountTag.init( dataFile, "AMNT_TAG" ) ;
}

void seekStatus(int rc, Str4ten &status)
{
    switch(rc)
    {
        case r4success:
            status.assign( "r4success" ) ;
            break;

        case r4eof:
            status.assign( "r4eof" ) ;
            break;

        case r4after:
            status.assign( "r4after" ) ;
            break;

        default:
            status.assign( "other" ) ;
            break;
    }
}

void printRecord( int rc )
{
    Date4 bDate ;
    Str4ten purchased ;
    Str4large name ;
    Str4ten status ;
    seekStatus( rc, status ) ;

    purchased.assignDouble( (double) amount, 8, 2 ) ;
    bDate.assign( birthDate ) ;
    name.assign( fName ) ;
    name.trim( ) ;
    name.add( " " ) ;
    name.add( lName ) ;
    cout << "Seek status : " << status.ptr( ) << endl ;
    cout << "--------------------------------" << endl ;

    cout << "Name     : " << name .ptr( ) << endl ;
    cout << "Address  : " << address.str( ) << endl ;
    cout << "Age : " << (int) age << " Married : "
                     << married.str( ) << endl ;
    cout << "Birth Date: "
                     << bDate.format( "MMMMMMMM DD, CCYY" ) << endl ;
    cout << "Comment: " << comment.str( ) << endl ;
    cout << "Purchased this year:$" << purchased.ptr( ) << endl ;
    getchar( ) ; // Pause for an Enter Key
}

void main( void )
{
    int rc;

    OpenDataFile( ) ;

    dataFile.select(  addressTag ) ;
    rc = dataFile.seek( "123 - 76 Ave   ") ;
    printRecord( rc ) ;

    rc = dataFile.seek( "12") ;
    printRecord( rc ) ;

    rc = dataFile.seek( "12             ") ;
    printRecord( rc ) ;

    dataFile.select( birthdateTag) ;
    rc = dataFile.seek( "19581111") ; // October 11, 1958
    printRecord( rc ) ;

    dataFile.select( amountTag) ;
    rc = dataFile.seek( 98.99) ;
    printRecord( rc ) ;

    rc = dataFile.seek( " 98.99") ;
    printRecord( rc ) ;

                                       //The following code finds all the occurrences
                                       //John Albridge in the tag
    dataFile.select( nameTag) ;
    rc = dataFile.seekNext( "Albridge  John      ") ;
    for (rc; rc == r4success; rc = dataFile.seekNext("Albridge  John      ") )
        printRecord( rc );

    codeBase.closeAll( ) ;
    codeBase.initUndo( ) ;
```

```
}
```

Performing Seeks

There are two functions in the CodeBase library that perform seeking: **Data4::seek** and **Data4::seekNext**. Both may seek for character, numeric or binary data.

Data4::seek with character data

Data4::seek accepts a **(char *)** string as the search key and performs a seek using the selected tag. If there is no selected tag, the first tag of the first opened index is used for the seek.

Code fragment from SEEKER.CPP

```
dataFile.select( addressTag ) ;
rc = dataFile.seek( "123 - 76 Ave   ") ;
printRecord( rc ) ;
```

Seeking on Date tags

When performing seeks using Date tags, **Data4::seek** must be passed an eight character string containing a date in the standard "CCYYMMDD" format. **Date4::ptr** is often used with **Data4::seek** to locate Date key values. See the "Date4 Class Functions" chapter for details on using the **Date4** functions.

```
dataFile.select( birthdateTag ) ;
rc = dataFile.seek( "19581111" ) ; // October 11, 1958
printRecord( rc ) ;
```

Seeking on Binary data

Use the **int Data4::seek(char *key, int len)** version to seek on character tags that contain binary data. This version of the seek function can be used to seek without regard for nulls, since *len* specifies the length of the search key.

Seeking on Numeric tags

When **Data4::seek** uses a Numeric tag, and the search key is a character pointer, the character array is converted into **(double)** value before searching. As a result, you do not have to worry about formatting the character parameter to match the key value.

```
rc = dataFile.seek( "98.99" ) ;
printRecord( rc ) ;
```

Data4::seek also accepts a **(double)** value. This version of seek should only be used on numeric keys. All numeric tags, regardless of the number of decimals, store their key entries as **(double)** values. When seeking on numeric tags, it is more efficient to seek with a **(double)** value than a character representation.

```
dataFile.select( amountTag ) ;
rc = dataFile.seek( 98.99 ) ;
printRecord( rc ) ;
```

Data4::seek returns an integer value indicating the success or failure in locating the key value in the selected tag.

Data4::seekNext

Data4::seekNext only differs from **Data4::seek** in that the search begins at the current position in the tag. This provides the capability of performing an incremental search through the data base. Consequently, all index keys matching the search key can be found by repeated calls to **Data4::seekNext**. On the other hand, **Data4::seek** always begins its search from the first logical record as determined by the selected tag and therefore, it only locates the first index key in the tag that matches the search key.

Data4::seekNext works in the following manner. If the index key at the current position in the tag does not match the search key, **Data4::seekNext** calls **Data4::seek** to find the first occurrence of the search key. Conversely, if the index key at the current position does match the search key, **Data4::seekNext** tries to find the next occurrence of the search key.

```
dataFile.select( nameTag) ;
rc = dataFile.seekNext( "Albridge  John      ") ;
for (rc; rc == r4success;
                 rc = dataFile.seekNext("Albridge  John      ") )
    printRecord( rc );
```

Exact Matches

CodeBase gives you the ability to perform both exact and partial matches on tags. An exact match occurs when the index key in the tag is the same as the search key. When searching using a character tag, the criteria that **Data4::seek** and **Data4::seekNext** use for determining an exact match is as follows:

- The characters in the search string must be the same as corresponding characters in the index key.
- The comparison is case sensitive. Therefore a search key of "aaaa" does not match an index key of "AAAA".

  To do a case insensitive search, one must change the indexing strategy, not the search method. Create a tag with the index expression "UPPER(fieldName)" and then a search for "AAAA" will be case insensitive. Therefore, both "Aaaa" and "aAaA" will be found.
- The strings are only compared up to end of the shortest value. Therefore a search key of "abcd" exactly matches an index key of "abcdefg" because only four characters are compared.

Note You can force CodeBase to compare additional characters by padding the search key out to the full length of the index key. Therefore a search key of "abcd   " does not exactly match "abcdefg".

When either **Data4::seek** or **Data4::seekNext** find an exact match, a value of **r4success** is returned.

Partial Matches

When a seek is performed, the tag is positioned to the place in the tag where the search key should be located. If this position lies between two index keys (ie. the tag was not exactly found), then a partial match has occurred. The record whose index key is directly after the "correct" position is loaded into the record buffer. When this happens, a value of **r4after** is returned by **Data4::seek** and **r4entry** or **r4after** is returned by **Data4::seekNext**. Consider the following three seeks.

Code fragment from SEEKER.CPP

```
dataFile.select( addressTag ) ;
rc = dataFile.seek( "123 - 76 Ave   ") ;
printRecord( rc ) ;

rc = dataFile.seek( "12") ;
printRecord( rc ) ;

rc = dataFile.seek( "12             ") ;
printRecord( rc ) ;
```

Assuming that there is a record in the data file containing "123 - 76 Ave" in the address field: The first time **Data4::seek** is called, it returns **r4success** because its search key exactly matched and was the same length as the index key. The second call will also return **r4success** because it only compares the first two characters. The third call however, returns **r4after** because it could not find an exact match. All three, however, result in the "123 - 76 Ave." record being loaded into the record buffer.

When **Data4::seekNext** is called and the index key at the current position in the tag does not match that of the search key, **Data4::seek** is called to find the first instance of the search key. Should this seek fail, **r4after** is returned and tag is positioned as discussed above.

On the other hand, if **Data4::seekNext** is called and index key at the current position does match the search key, **Data4::seekNext** searches for the next occurrence of the search key. If this seek fails, **r4entry** is returned and the tag is positioned in the same manner as **r4after**.

No Match

If there are no index keys in the tag or if the search key's position is after the last index key, then **Data4::seek** or **Data4::seekNext** return **r4eof**.

Seeking On Compound Keys

Performing seeks on compound index keys is slightly more difficult. Before a seek can be performed on a tag with a compound index expression, a search key must be created in a similar manner.   For example, if the index expression for the tag is "L_NAME + F_NAME", an example search key can be built as follows:

```
Field4 fname( data, "FNAME" ), lname( data, "LNAME") ;
Str4large fullName ;
fullName.setLen( fname.len( ) + lname.len( ) ) ;

fullName.set( ' ' ) ;
fullname.replace( Str4ptr( "Aldbridge" ) ) ;
fullname.replace( Str4ptr( "John" ), fname.len( ) ) ;
```

The first part of the search key, "Albridge   ", matches the format of field L_NAME and is padded out with blanks to the length of that field. The second part, "John   ", matches field F_NAME.

Note

The various parts of the search key must be padded out to the full length of their respective fields. Failure to do so may cause an incorrect seek.

Building a compound key with the string classes

The string classes may be used to build a compound effectively by using **Str4::setLen**, **Str4::set, Str4::add,** and **Str4::replace** to properly position the various parts of the key.

```
Str4large fullName ;
Field4 lastName( dataFile, "L_NAME" ) ;

fullName.setLen( lastName.len( ) ) ;
fullName.set( ' ' ) ;
fullName.replace( Str4ptr( "Albridge" ) ) ;

fullName.add( "John      " ) ;
```

A similar strategy is used when seeking on compound index keys built with fields of different types. If the index expression is "PRODUCT + STR(COST,7,2)" where 'PRODUCT' is a Character field of length eight and Cost is a Numeric field, then a valid search key is "Dunsel 1234.21". Since the STR() function converts Numeric values to Character values, the search string is constructed with the first eight characters matching the PRODUCT field and the "1234.21" matching the return value of STR().

Code Fragment to build "Dunsel 1234.21"

```
Str4large searchKey ;
searchKey.assignDouble( 1234.21, 7, 2 ) ;
// searchKey contains "1234.21"
searchKey.insert( Str4ptr("Dunsel  ") ) ; // length of 8
//searchKey contains "Dunsel  1234.21"
```

Note

You should note the difference between the dBASE "+" and "-" operators when they are applied to Character values. The "+" includes any trailing blanks when the Character values are concatenated. For example if the index expression is "L_NAME + F_NAME", the search key should be "Krammer   David   ". The "-" removes any trailing blanks from the first string. The second string is concatenated and the previously removed blanks are added to the end of the concatenated string. As a result, if the index expression is "L_NAME - F_NAME", the search key should be "KrammerDavid   ".

Group Files

In Clipper, the .**NTX** indexes must be manually opened each and every time the data file is opened. This causes programming time to be increased, as well as having a good chance of forgetting to open an index file. dBASE IV and FoxPro use compound index files (more than one tag in a file) and production index files (automatically opens when data file opens) to avoid these problems.

CodeBase has introduced group files in order to compensate for this limitation of the .**NTX** file format. A group file allows you to use the same function calls when using .**NTX** index files as you would when using .**CDX** and .**MDX** index files.

This is accomplished by emulating production indexes and multiple tags per index file. The same database that was shown in Figure 5.2 is depicted in Figure 5.5 using .**NTX** index files and group files.

Creating Group Files

A group file is automatically created when using **Index4::create** or **Data4::create** when using **.NTX** index compatibility. In addition, a group file can be created for existing index files using a text editor. Simply create a file with a file name extension of ".CGP". Then you enter the names of the index files to be grouped together. Enter one index file name per line.

GROUP FILE PRODUCTS.CGP

For example, here is the actual contents of group file 'PRODUCTS.CGP' from Figure 5.5:

```
PRO_TAG
P_NUM_TAG
```

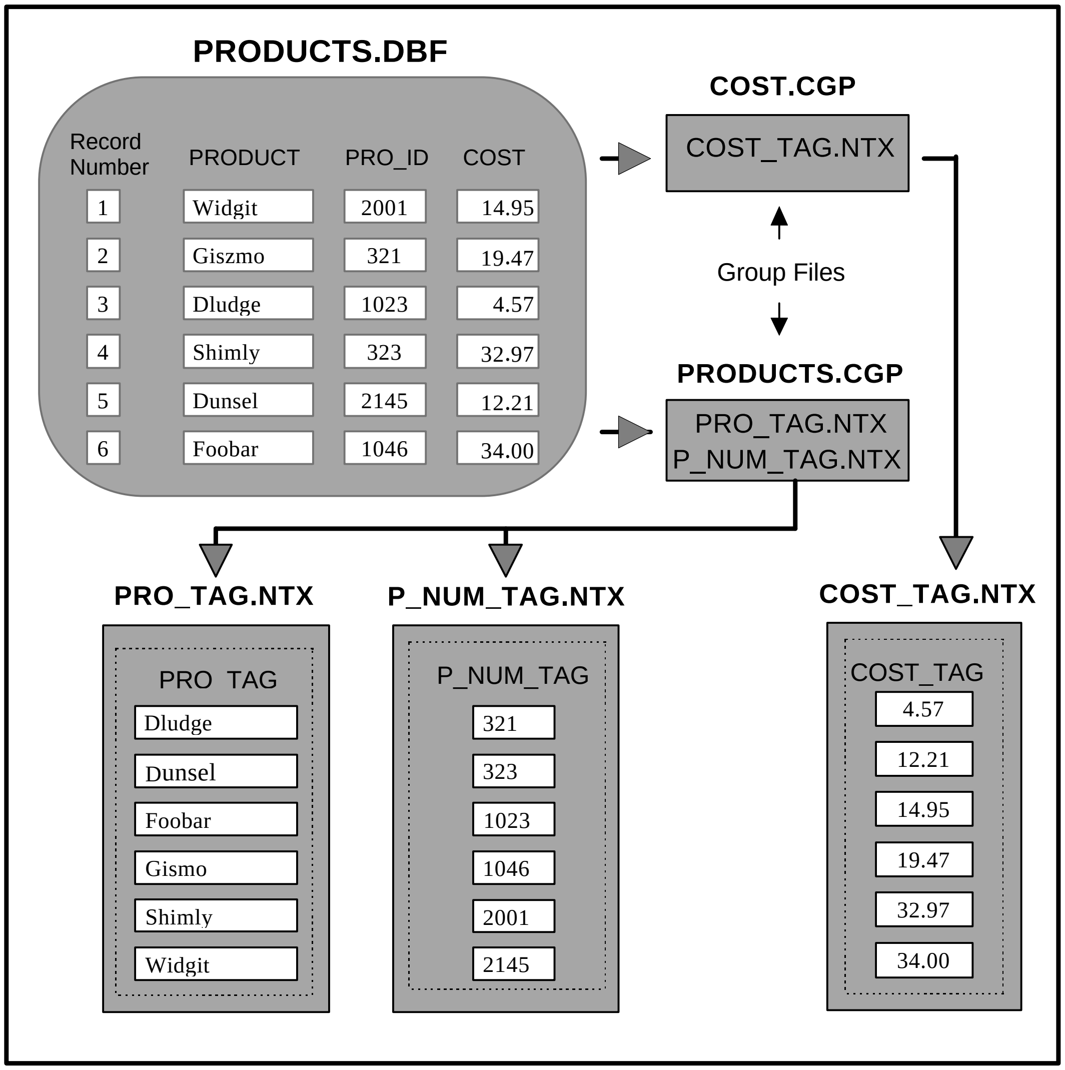


Figure 5.5 Group Files

Bypassing Group Files

Because group files are unique to CodeBase, index files generated with Clipper do not have group files. Under these conditions, it may be more convenient for you to bypass the group files and access the .**NTX** group files directly instead of creating your own group files.

Disabling The Auto Open

When using **.NTX** index compatibility, CodeBase automatically tries to open a group file when a data file is opened. If you have tried to open a data file that does not have a group file, you will encounter the following error:

```
ERROR #  -60
UNABLE TO OPEN  DATA1.CGP
PRESS ANY KEY TO CONTINUE
```

This results from CodeBase being unable to locate a group file. This behavior can be disabled by setting the **Code4::autoOpen** flag to false (zero).

Creating Index Files

You can create .**NTX** files without a corresponding group file by passing **Index4::create** a null file name. As documented earlier, the tag names also become file names.

PROGRAM NOGROUP.CPP

This program demonstrates how .**NTX** index files can be created without group files.

```
#include "d4all.hpp"
extern unsigned _stklen = 10000 ; // for all Borland compilers

Code4  codeBase ;
Data4  dataFile ;
Tag4   nameTag, addressTag, ageTag, dateTag ;

TAG4INFO tagInfo[] =
{
    {"NAME","L_NAME+F_NAME",0,0,0},
    {"ADDRESS","ADDRESS ",0,0,0},
    {"AGE_TAG","AGE",0,0,0},
    {"DATE","BIRTH_DATE",0,0,0},
    {0,0,0,0,0}
} ;

void  main ( void )
{
    codeBase.autoOpen = 0;
    codeBase.safety = 0;

    dataFile.open( codeBase, "DATA1.DBF" ) ;
    Index4 index ;
    index.create( dataFile, 0, tagInfo ) ;
    nameTag.init( dataFile, "NAME" ) ;
    addressTag.init( dataFile, "ADDRESS" ) ;
    ageTag.init( dataFile, "AGE_TAG" ) ;
    dateTag.init(dataFile, "DATE" );

    codeBase.closeAll( ) ;
    codeBase.initUnd o( ) ;
}
```

Opening Index Files

A single .**NTX** index file can also be opened without using group files.

WARNING

The following example program contains code segments that are specific to CodeBase libraries built with.**NTX** index file compatibility.

PROGRAM NOGROUP2.CPP

This program demonstrates how .**NTX** index files can be opened without using a group file.

```
#include "d4all.hpp"
extern unsigned _stklen = 10000 ; // for all Borland compilers

Code4   codeBase ;
Data4   dataFile ;
Field4  fName, lName, address, age, birthDate, married, amount ;
Field4memo comment ;

Tag4    nameTag, addressTag, ageTag, dateTag;

void printRecords( void )
{
    ...
}

void main ( void )
{
    codeBase.autoOpen = 0;
    codeBase.safety = 0;

    dataFile.open( codeBase, "DATA1.DBF" ) ;

    nameTag.open( dataFile, "NAME" ) ;
    addressTag.open( dataFile, "ADDRESS" ) ;
    ageTag.open( dataFile,  "AGE_TAG" ) ;
    dateTag.open( dataFile, "DATE" ) ;

    dataFile.select( nameTag ) ;
    printRecords();

    codeBase.closeAll( ) ;
    codeBase.initUndo( ) ;
}
```

The function that provides this functionality is **Tag4::open**. It opens the specified file, stores its information in the **Data4** object, and initializes the **Tag4** object to its tag.

Code fragment from NOGROUP2.CPP

```
nameTag.open( dataFile, "NAME" ) ;
addressTag.open( dataFile, "ADDRESS" ) ;
ageTag.open( dataFile, "AGE_TAG" ) ;
dateTag.open( dataFile, "DATE" ) ;
```

If a tag opened with **Tag4::open** is to be associated with a specific **Index4** object, it may be provided as a second parameter. In general, however, this added functionality is unneeded.

Alternatively, **Index4::Index4** or **Index4::open** may be used to open an **.NTX** index file. When doing so, the**.NTX** file name extension must be provided.

Reindexing

Index files become out of date if they remain closed when their data file is modified. Alternatively, a computer could be turned off in the middle of an update. When an index is out of date, the data file may contain records that do not have corresponding tag entries, or the index file may have tag entries that specify the wrong record.

To rectify this problem, it is sometimes necessary to reindex the index files. You could accomplish this in two ways. First you could delete the index file from your system and then recreate the index, or you could open an existing index file and call one of the reindex functions.

Reindexing All Index Files

If you suspect one or all of your index files may be out of date, you can use the **Data4::reindex** function. This function reindexes any index files that are associated with that data file and are currently open. This includes both production and non-production index files.

```
    dataFile.reindex( ) ;
```

Reindexing A Single Index

If you have several index files but only one is out of date, you can reindex just that index file by using the **Index4::reindex** function. This function reindexes the index file for the **Index4** object. All tags associated with the **Index4** object are automatically updated.

```
    Index4 indexFile ;
    indexFile = dataFile.index( "INDEX" ) ;
    indexFile.reindex( ) ;
```

You can initialize the **Index4** object with either **Index4::open** or **Index4::create**, or by using **Data4::index** with the index file's name.

Advanced Topics

The next section contains details on copying the index file structures.

Copying Index File Structures

You may occasionally require a new index file that has an identical structure as an existing index file. This situation usually arises when you are making a copy of a data file.

As a parallel to **Field4info** class, CodeBase provides the **Tag4info** class. A **Tag4info** constructor can be used to copy the structure of an index file. Once constructed, the original index file may be closed, if necessary, and the new index file may be created with **Data4::create** or **Index4::create**. You can pass the array returned from **Tag4info::tags** to the **Data4::create** or **Index4::create** to create a new index file.

The **TAG4INFO** stored in the **Tag4info** class is allocated dynamically and should therefore be deallocated after you are finished using it. **Tag4info::free** may be called to free up the memory, otherwise the **Tag4info** destructor may be called implicitly.

PROGRAM COPYDAT2.CPP

Program COPYDAT2.CPP takes two file names as arguments. It uses the structure from the data file specified by the first argument to create an empty data file with the name specified by the second argument. A duplicate index file is made if one exists.

```
#include "d4all.hpp"
extern unsigned _stklen = 10000 ; // for all Borland compilers

void main(int argc,char *argv[])
{
    if(argc != 3)
    {
        printf("USAGE:COPYDATA <FROMFILE> <TOFILE>");
        exit(1);
    }

    Code4   codeBase;

    codeBase.safety = 0;
    Data4 dataFile( codeBase, argv[1] ) ;
    codeBase.exitTest( ) ;

    Field4info fields( dataFile ) ; // copy the fields

    // obtain the Index4 object of the p roduction index, if one exists
    Index4 index = dataFile.index( dataFile.alias( ) ) ;
    Data4 dataCopy ;

    if( index.isValid( ) )
    {
        Tag4info tags( index ) ;
        dataCopy.create( codeBase, argv[2], fields.fields( ), tags.tags( ) ) ;
        tags.free( ) ;
    }
    else
        dataCopy.create( codeBase, argv[2], fields.fields( ) ) ;

    codeBase.closeAll( ) ;
    fields.free( ) ;
    codeBase.initUndo( ) ;
}
```

6 Queries And Relations

One of the most powerful features of CodeBase is its relation and query capabilities. By using relations, you can turn a collection of separate data files into a fully relational database. Relations are used for two tasks: lookups and queries.

Lookups are used to automatically locate records between related data files. Essentially, this makes several related data files act like a single large data file.

Queries retrieve groupings of records from the database that meet conditions that you can specify at run-time. Queries use CodeBase's Query Optimization, which greatly reduces the time it takes to perform queries.

Relations

In its simplest form, a relation is a connection between two data files that specifies how the records from the first data file are used to locate one or more records from the second.

Master & Slave Data File

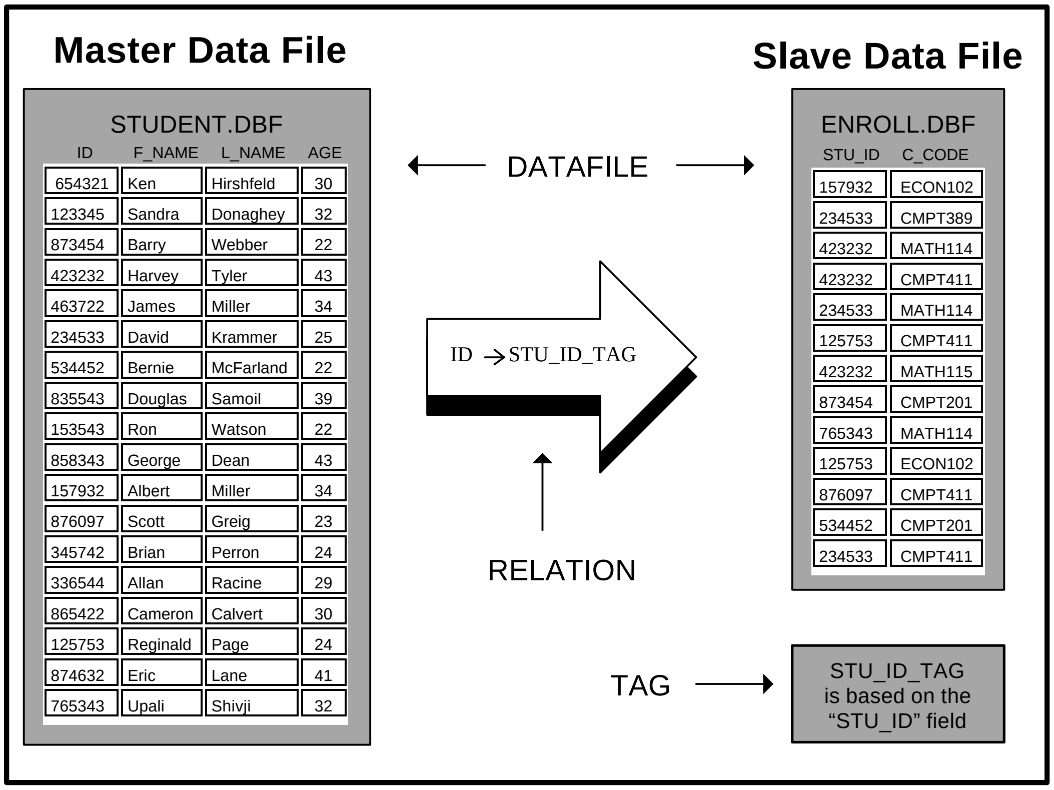
The data file which controls the others is called the master. The controlled data files are called slaves. A master can have any number of slaves which in turn can be masters of other slaves.

Master Expression & Slave Tag

To create a relation, you usually provide two pieces of information. The first is the *master expression*, and the second is a tag based on the slave data file called the *slave tag*.

The purpose of the master expression is to generate an index key called the *lookup key*. The master expression is evaluated for each record in the master data file and the resulting value is that record's lookup key. A record in the slave is then found by performing a seek using the lookup key on the slave tag.

The most common type of master expressions are ones which only contain the name of a field from the master data file. The slave tag ordering is usually based on an identical field in the slave data file. Figure 6.1 shows a relation of this type.



STUDENT.DBF

ID	F_NAME	L_NAME	AGE
654321	Ken	Hirshfeld	30
123345	Sandra	Donaghey	32
873454	Barry	Webber	22
423232	Harvey	Tyler	43
463722	James	Miller	34
234533	David	Krammer	25
534452	Bernie	McFarland	22
835543	Douglas	Samoil	39
153543	Ron	Watson	22
858343	George	Dean	43
157932	Albert	Miller	34
876097	Scott	Greig	23
345742	Brian	Perron	24
336544	Allan	Racine	29
865422	Cameron	Calvert	30
125753	Reginald	Page	24
874632	Eric	Lane	41
765343	Upali	Shivji	32

ENROLL.DBF

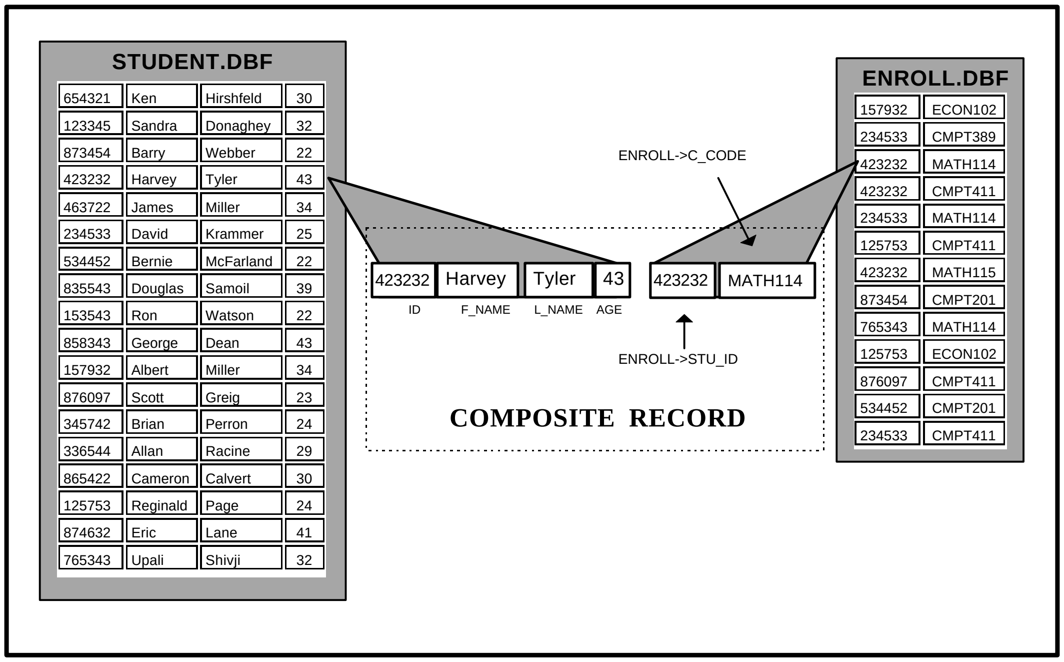
STU_ID	C_CODE
157932	ECON102
234533	CMPT389
423232	MATH114
423232	CMPT411
234533	MATH114
125753	CMPT411
423232	MATH115
873454	CMPT201
765343	MATH114
125753	ECON102
876097	CMPT411
534452	CMPT201
234533	CMPT411

Figure 6.1 Relation on two data files

In this example, the master expression is "ID" and the slave tag is the tag STU_ID_TAG. The index expression for the slave tag is "STU_ID". If, for example, the current record for the master data file is set to record number 4, the master expression would then evaluate to "423232". The lookup on the slave data is then performed, locating record number 3.

Composite Records

Related records from the master and slaves data files are collectively referred to as a *composite record*. The composite record consists of the fields from the master along with the fields from the slave data files. The following is a composite record from the previous relation example:



STUDENT.DBF

654321	Ken	Hirshfeld	30
123345	Sandra	Donaghey	32
873454	Barry	Webber	22
423232	Harvey	Tyler	43
463722	James	Miller	34
234533	David	Krammer	25
534452	Bernie	McFarland	22
835543	Douglas	Samoil	39
153543	Ron	Watson	22
858343	George	Dean	43
157932	Albert	Miller	34
876097	Scott	Greig	23
345742	Brian	Perron	24
336544	Allan	Racine	29
865422	Cameron	Calvert	30
125753	Reginald	Page	24
874632	Eric	Lane	41
765343	Upali	Shivji	32

ENROLL.DBF

157932	ECON102
234533	CMPT389
423232	MATH114
423232	CMPT411
234533	MATH114
125753	CMPT411
423232	MATH115
873454	CMPT201
765343	MATH114
125753	ECON102
876097	CMPT411
534452	CMPT201
234533	CMPT411

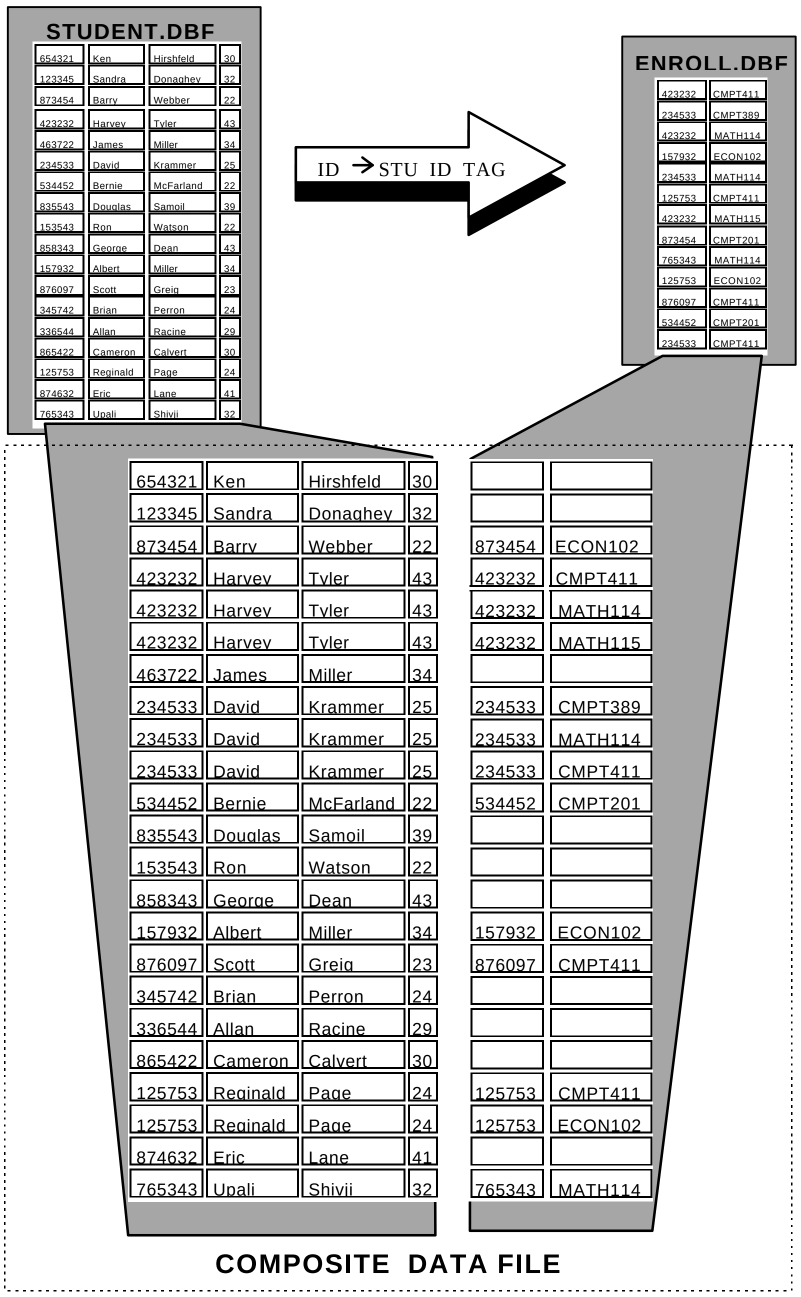
Figure 6.2 A Composite Record

When using fields from the composite record in a dBASE expression, you must precede any field names from slave data files with the field name qualifier. The field name qualifier is the name of the data file followed by the "->" characters. This avoids any potential naming conflicts caused by data files with field names in common.

For example, " ENROLL->STU_ID = 876987" is a dBASE expression containing fields from the slave data file. This expression could be used as part of a query (discussed later in this chapter).

Composite Data Files

The composite data file is the set of composite records that are produced by the relation. Figure 6.3 lists the composite data file for the example relation.



STUDENT.DBF

654321	Ken	Hirshfeld	30
123345	Sandra	Donaghey	32
873454	Barry	Webber	22
423232	Harvey	Tyler	43
463722	James	Miller	34
234533	David	Krammer	25
534452	Bernie	McFarland	22
835543	Douglas	Samoil	39
153543	Ron	Watson	22
858343	George	Dean	43
157932	Albert	Miller	34
876097	Scott	Greig	23
345742	Brian	Perron	24
336544	Allan	Racine	29
865422	Cameron	Calvert	30
125753	Reginald	Page	24
874632	Eric	Lane	41
765343	Upali	Shivji	32

ENROLL.DBF

423232	CMPT411
234533	CMPT389
423232	MATH114
157932	ECON102
234533	MATH114
125753	CMPT411
423232	MATH115
873454	CMPT201
765343	MATH114
125753	ECON102
876097	CMPT411
534452	CMPT201
234533	CMPT411

COMPOSITE DATA FILE

654321	Ken	Hirshfeld	30		
123345	Sandra	Donaghey	32		
873454	Barry	Webber	22	873454	ECON102
423232	Harvey	Tyler	43	423232	CMPT411
423232	Harvey	Tyler	43	423232	MATH114
423232	Harvey	Tyler	43	423232	MATH115
463722	James	Miller	34		
234533	David	Krammer	25	234533	CMPT389
234533	David	Krammer	25	234533	MATH114
234533	David	Krammer	25	234533	CMPT411
534452	Bernie	McFarland	22	534452	CMPT201
835543	Douglas	Samoil	39		
153543	Ron	Watson	22		
858343	George	Dean	43		
157932	Albert	Miller	34	157932	ECON102
876097	Scott	Greig	23	876097	CMPT411
345742	Brian	Perron	24		
336544	Allan	Racine	29		
865422	Cameron	Calvert	30		
125753	Reginald	Page	24	125753	CMPT411
125753	Reginald	Page	24	125753	ECON102
874632	Eric	Lane	41		
765343	Upali	Shivji	32	765343	MATH114

Figure 6.3 The Composite Data File

The composite data file does not physically exist on disk. It is merely a way of viewing the information in the data files of the relation.

For example, when the relation is applied to record number 3 of the master data file the following composite record is generated:

Composite Data File Record # 3

873454	Barry	Webber	22	873454	CMPT201

By default, if the relation does not find a match in the slave data file, the slave's fields in the composite record are left as blanks. For example record number 1 in the master data file does not have a corresponding slave record, so the generated composite record is:

Composite Data File Record # 1

654321	Ken	Hirshfield	30		

Complex Relations

Although having a relation between two data files is quite useful, it is often necessary to set relations between one master and several slaves, or to set relations between the slaves and other data files.

Single Master, Multiple Slave Relations

Relations with one master and several slaves are very similar to the single master, single slave relations. In this case there is a master expression and slave tag for each slave, and the composite record consists of fields from all the data files. CodeBase permits any number of slaves for a single master. The only limitations are the resources of your system.

Multi-Layered Relations

There are many situations that require a master with a relation to a slave, which in turn has a relation to other data files. CodeBase supports an unlimited number of these multi-layered relations and will automatically perform lookups in any associated slave data files.

The Top Master

A master data file is a master only in the context of a specific relation. The data file can also be slave in a different relation. If the data file is not a slave in any other relation, it is called a *top master*.

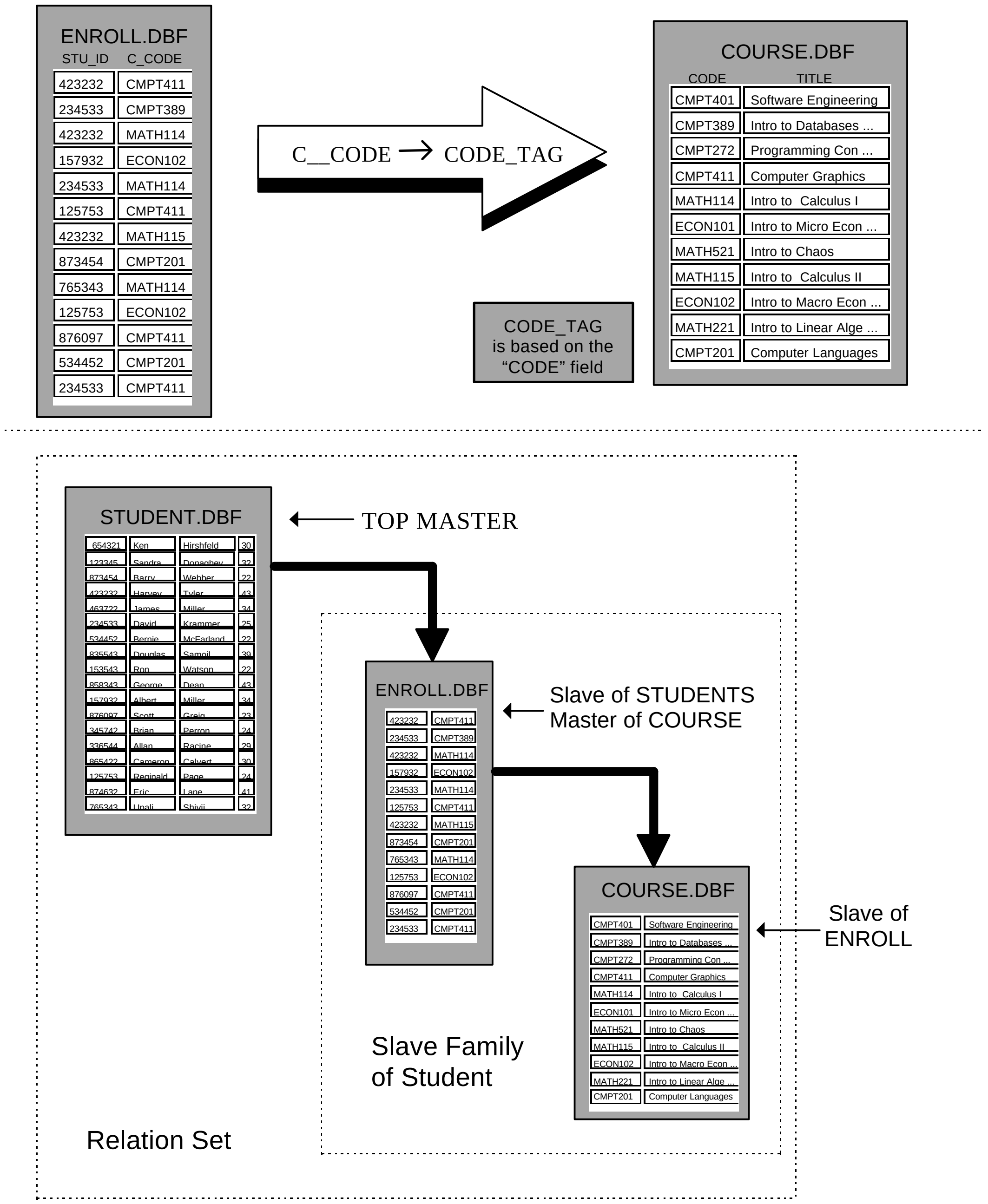


Figure 6.4 A Relation Set

The Relation Set

A relation set consists of a top master and all other connected relations. There must be exactly one top master in a relation set.

Figure 6.4 describes a new relation between the ENROLL.DBF data file and the new data file COURSE.DBF. It shows the entire relation set and points out the various relations between the data files.

Relation Types

There are three basic types of relations. They are *exact match*, *scan* and *approximate match* relations. The type of relation determines what record or records are located during a lookup.

Exact Match Relations

An *exact match* relation permits only a single match between the master and slave data files. Both one to one and many to one relations are exact match relations. In a one to one relation, there is only one record in the master that matches a single record in the slave. In a many to one relation, there are one or more records in the master that match a single slave record.

If there are multiple slave records that match a single master record, then a composite record is only generated for the first slave record, as illustrated in Figure 6.5.

There are three records in the slave data file that match record number 4 of the master. Because this is an exact relation a composite record is made from only the first matching record of the slave data file.

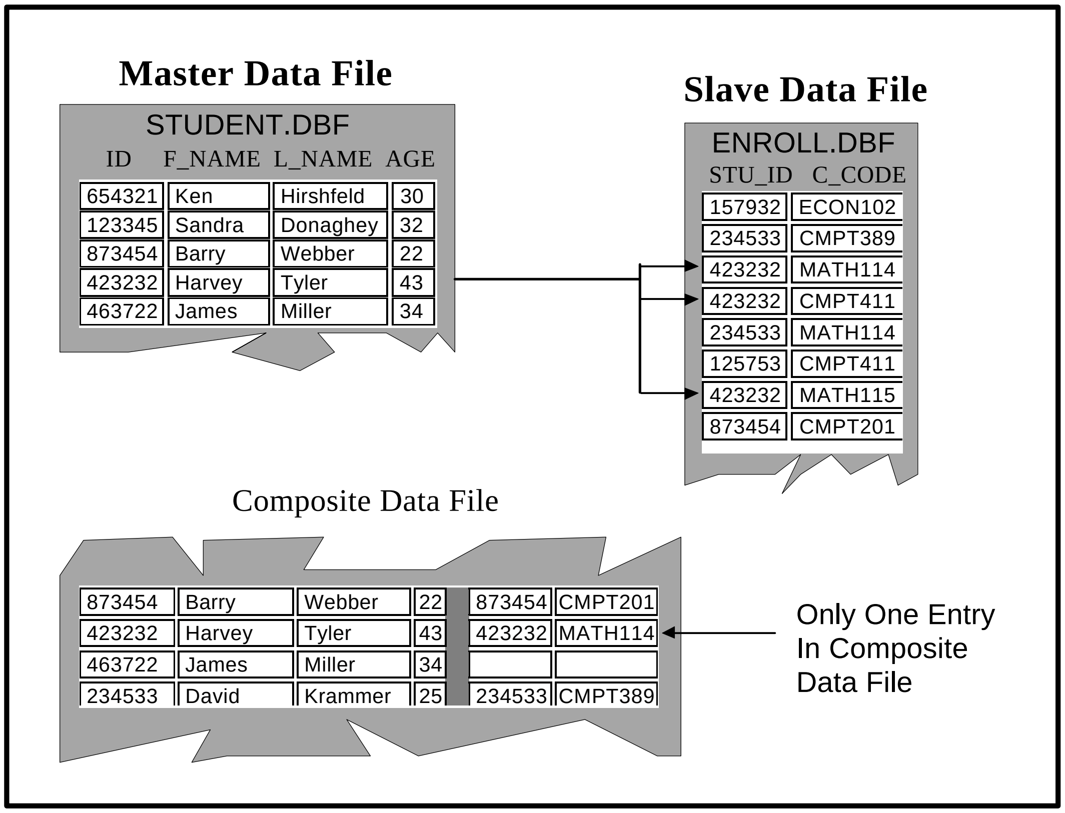


Figure 6.5 An Exact Match Relation

Scan Relations

In a *scan* relation, if there are multiple slave records for a master record, there is one composite record in the extended data file for each of the matching slave records. Both one to many and many to many relations are scan relations. In a one to many relation, each record in the master may have multiple matching slave records. A many to many relation is the same as one to many except that different master records may match the same slave record.

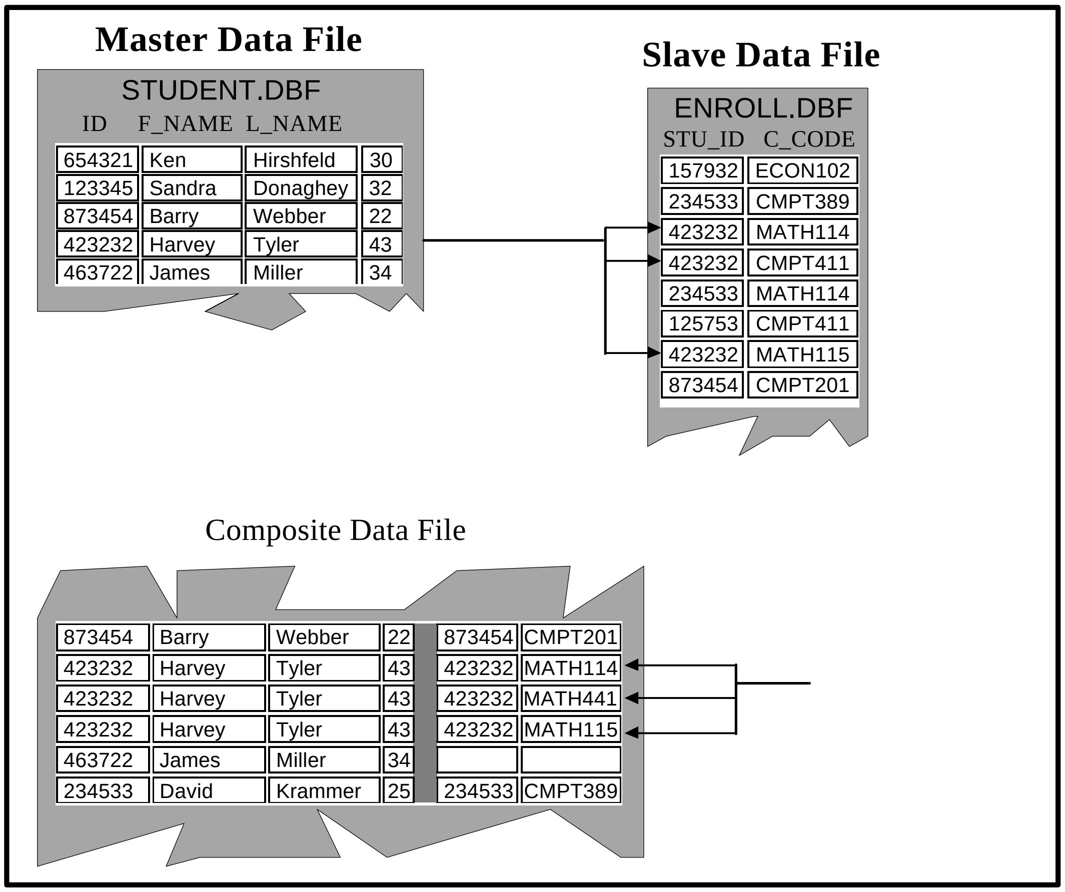


Figure 6.6 A Scan Relation

Approximate Match Relations

The final type of relation is the approximate match relation. This is similar to the exact match relation in that it permits only one match for a master record. The only difference is the way it behaves when an exact match is not found in the slave data file. If the match fails, the slave record whose index key appears next in the slave tag is used instead. Approximate match relations are generally quite rare and are usually used only when a range of records are represented by a single high value.

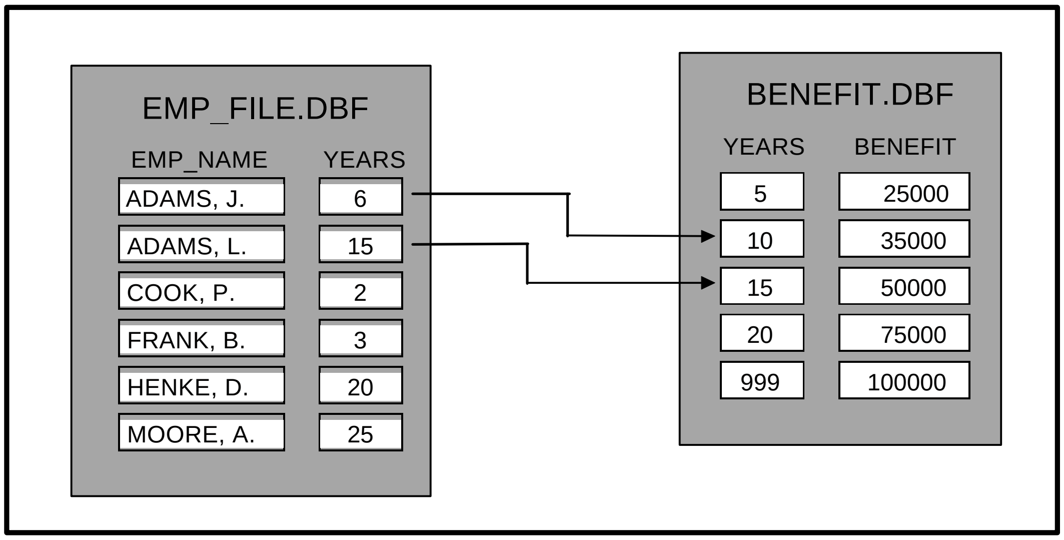


Figure 6.7 Approximate Match Relations

An approximate match relation is shown in Figure 6.7. In this case, the employees' retirement benefits are determined by the number of years that they put into the company. Instead of making an entry for each possible year served, the BENEFIT.DBF only lists the upper limit for each pay out level. The first pay out level is from zero to five years, the second is six to ten, et cetera until the maximum entry of twenty-one years and above pays out 100000.

Creating Relations

Relations are created during the execution of your application. Before you can create a relation, you must have opened all the data files and any index files that are used in the relation set.

PROGRAM RELATE1.CPP

Program RELATE1.CPP sets up a scan type relation between the STUDENT.DBF and ENROLL.DBF data files that were illustrated in Figure 6.1. This program lists all of the records in composite data file.

```
#include "d4all.hpp"
extern unsigned _stklen = 10000;

Code4      codeBase( 0 ) ;
Data4      student, enrolment ;
Relate4set master ;
Tag4       idTag, nameTag ;

void openDataFiles( void ) ;
void setRelation( void ) ;
void printRecord( void ) ;

void openDataFiles( )
{
    codeBase.init( ) ;

    student.open( codeBase,"STUDENT" ) ;
    enrolment.open( codeBase,"ENROLL");

    nameTag.init( student,"NAME");
    idTag.init( enrolment,"STU_ID_TAG");

    codeBase.exitTest( ) ;
}

void setRelation()
{
    master.init( student ) ;
    if( ! master.isValid( ) )
        exit( 1 ) ;

    Relate4 slave( master, enrolment, "ID", idTag) ;
    slave.type( relate4scan ) ;
    master.top( ) ;

    // 'slave' falls out of scope, but the relation remains valid
}

void printRecord()
{
    Relate4iterator relation( master ) ;
    Data4   data ;
    Field4memo  field;

    for( relation; relation.isValid( ); relation.next( ) )
    {
        data = relation.data( ) ;

        for( int j = 1; j <= data.numFields( ); j++ )
        {
            field.init( data, j ) ;
            cout << field.str( ) << " " ;
        }
    }
    cout << endl ;
}

void listRecords( void )
{
    int rc;

    for(rc = master.top( ); rc != r4eof; rc = master.skip( ) )
        printRecord( ) ;

    cout << endl ;
    codeBase.unlock( ) ;
}

void main( void )
{
    openDataFiles( ) ;

    setRelation( ) ;

     listRecords( ) ;

     master.free( ) ;

    codeBase.closeAll( ) ;
    codeBase.initUndo( ) ;
}
```

The Relate4 class

The **Relate4** class contains the control information about a single data file in a relation. Every data file that is in the relation set must construct a **Relate4** object describing its placement in the relation. A relation's **Relate4** object contains information such as what data file is the master of which data file and what type of relation exists between them. In a single relation set, a data file may only be a slave to a single data file. That is, in any one relation set, a data file may only be used once. A data file may not be related to itself, nor may it be related to two different master data files.

Specifying The Top Master

The first step for creating a relation entails specifying which data file will be the top master of the relation set. This step is accomplished by constructing a **Relate4set** object.

Code fragment from RELATE1.CPP

```
Relate4set master ;
. . .
master.init( student ) ;
```

At this point, you have a complete relation set consisting of one data file -- the top master. Why would you want a relation set with just a single data file in it? A single data file relation set may be used when a query involves only the one data file.

Adding Slave Data Files

Adding slave data files to a relation is accomplished by constructing (or initializing) a **Relate4** object. To do this, there are four pieces of information that you must specify.

First, the master data file's **Relate4** object must be provided. This can either be the **Relate4set** object for the relation set, or another constructed, valid **Relate4** object. The second piece of information is the slave's **Data4** object.

WARNING

A data file should only be used once in a relation set. The result of using a data file more than once in a relation set is undefined and unpredictable.

The master expression is the third piece of information needed when adding a slave. This dBASE expression is evaluated using the current record of the master data file to produce a lookup key, for each lookup performed.

The final piece of information is a **Tag4** object for one of the slave's tags. During a lookup, a search of the lookup key (generated by the master expression) is made using this tag.

When the **Relate4::Relate4** constructor is invoked, it adds the slave data file to the relation and constructs a **Relate4** object. If the new relation has no lower relations and the relation is an exact match (the default), **Relate4::Relate4** may be invoked without constructing an object. This is done by invoking the constructor without an object name.

```
Relate4set master( student ) ;

Relate4( master, enrollment, "ID", idTag ) ;
```

You should construct an object if you plan on changing the default relation type or adding lower level relations.

```
Relate4 slave( master, enrollment, "ID", idTag) ;
slave.type( relate4scan ) ;
```

In the above code fragment, *master* is the **Relate4set** object for the top master data file and *enrollment* is the **Data4** object of a slave data file. The master expression is "ID". As a result, the lookup key is the contents of the ID field from the master data file. The lookups are performed using a tag from the slave data file. This tag is identified by *idTag*.

Creating Slaves Without Tags

CodeBase allows an alternative method of performing lookups that does not use tags from the slave data file. Instead of having the master expression evaluate to a lookup key, it evaluates to a record number. This record number specifies the physical record number of the slave record retrieved from the slave data file.

To use this method, simply pass an uninitialized **Tag4** object for the last parameter of **Relate4::Relate4** or **Relate4::init**. This method is mainly useful on static unchanging slave data files that are never packed. This method has the advantage of being faster and more efficient than performing seeks on a tag. The following is a example of this method.

```
 Relate4 slave( master, slave,"SLAVE_REC", Tag4( ) );
```

This example assumes that the master data base has a "SLAVE_REC" field which contains the physical record number of records in the slave data file. It would be the responsibility of the programmer to ensure that the record number stored in the "SLAVE_REC" field would reference an appropriate record for the master data file record.

Note

If the master expression contains a reference to a non existent record number (<= 0 or > **Data4::recCount**), CodeBase generates a -70 error ("Reading File", see "Appendix A: Error Codes" in the *Reference Guide*).

Setting The Relation Type

The default type of relation is an exact match relation. You can change a relation's type by calling **Relate4::type**. This function takes an integer specifying the type of the relation. The valid integer values are specified by the following defined CodeBase constants:

- **relate4exact** (Default) This specifies an exact match relation.
- **relate4approx** This specifies an approximate match relation.
- **relate4scan** This specifies a scan relation.

Setting The Error Action

Sometimes a record in the slave data file cannot be located from a master data file record. This occurs when you are using an exact type relation and there is no match for the lookup key, or when the relation is using direct record lookup and the generated record number does not exist in the slave data file. Under either of these circumstances, there are several ways in which CodeBase can react. **Relate4::errorAction** specifies which method is used. The valid error action codes are as follows:

- **relate4blank** (Default) This means that when a slave record cannot be located, it becomes a blank record.
- **relate4skipRec** This code means that the entire composite record is skipped as if it did not exist.
- **relate4terminate** This means that a CodeBase error is generated and the CodeBase relate function, usually **Relate4set::skip**, returns an error code.

Moving Through The Composite Data File

CodeBase provides three functions for moving through the records in the composite data file. They are **Relate4set::top**, **Relate4set::bottom** and **Relate4set::skip**. These functions are used mainly for obtaining composite records when a query is performed.

Listing The Composite Data File

When used together, these three functions allow you iterate through all of the composite data file. The following code segment demonstrates this process:

```
void listRecords( void )
{
    int rc;
    for(rc = master.top( ); rc != r4eof; rc = master.skip( ) )
        printRecord( ) ;

    cout << endl ;
}
```

Finding the first record in the Composite data file

Relate4set::top sets the current record of the master data file to the first record in the composite data file. If there are slaves in the masters slave family, lookups are automatically performed. If there are no composite records in the composite data file, **r4eof** is returned.

Skipping through records

For each iteration of the **for** loop, **Relate4set::skip** is used to move to the next record in the composite data file. Just as in the **Data4::skip** function, **Relate4set::skip's** parameter specifies the number of records to be skipped. When there are no more records in the composite data file, **r4eof** is returned.

Note

Before **Relate4set::skip** is used, a call to **Relate4set::top** or **Relate4set::bottom** must be made.

Skipping backwards through the composite data file is also permitted. Since skipping backwards requires extra internal overhead, you must either call **Relate4set::bottom** or **Relate4set::skipEnable** to initialize the backwards skipping abilities.

An example of this is given in the following code segment:

```
for( int rc = master.bottom( ); rc != bof ;
                                            rc = master.skip( -1 ) )
```

Queries And Sorting

Once a relation has been set up, you can then perform queries upon the composite data file. A *query* allows you to retrieve selected records from the composite data file by specifying a query expression. Essentially the query specifies a subset of the composite data file, which is called the *query set*, by filtering out any composite records that do not meet the search criterion.

In addition to performing queries, you can also specify the order in which the composite records are presented.

When a query is specified, CodeBase uses its Query Optimization, whenever possible, to minimize the number of necessary disk accesses. This greatly improves performance when retrieving composite records from the query set.

PROGRAM RELATE2.CPP

Program RELATE2.CPP demonstrates how queries can be performed on a single data file.

```
#include "d4all.hpp"
extern unsigned _stklen = 10000;

Code4   codeBase ;
Data4   student ;

void openDataFiles()
{
    student.open( codeBase, "STUDENT" ) ;
    codeBase.exitTest( ) ;
}

void printRecord(Data4 dataFile )
{
    for( int j=1; j <= dataFile.numFields( ); j++ )
        cout << Field4memo( dataFile, j ).str( ) << " " ;

    cout << endl ;
}

void query( Data4 dataFile, char *expr, char *order = NULL )
{
    Relate4set relation( dataFile ) ;
    if( ! relation.isValid( ) ) codeBase.exit( ) ;

    relation.querySet( expr ) ;
    relation.sortSet( order ) ;

    for( int rc = relati on.top( ); rc != r4eof; rc = relation.skip( ) )
        printRecord( dataFile ) ;
    cout << endl ;

    codeBase.unlock( ) ;
    relation.free( ) ;
}

void main( void )
{
    openDataFiles( ) ;

    query( student, "AGE > 30" ) ;

    query( student, "UPPER( L_NAME) = 'MILLER'", "L_NAME + F_NAME" ) ;
    codeBase.closeAll( ) ;
    codeBase.initUndo( ) ;
}
```

The Query Expression

To generate a query, all you need to provide is a dBASE expression, called the *query expression*, which evaluates to a Logical value. This expression is used to determine whether a composite record should be included in the query. If the query expression evaluates to .TRUE. then the record is kept, otherwise it is discarded from the query.

Setting the query expression

The query expression is specified for a relation set by the **Relate4set::querySet** function.

The query expression can reference fields from different files in the relation set. When a query is made, the default data base used is the master. This means that the field will automatically be associated with the master data base. To refer fields of different data files use the file's alias followed by an arrow (->) and the field name.

For example, suppose a relation consists of a master data file called FATHER.DBF and a slave data file called SON.DBF. Assume that each file has a field called NAME.

For these data files, the following queries are equivalent:

```
relation.querySet( "FATHER->NAME = 'Ben Nyland'") ;
relation.querySet( "NAME = 'Ben Nyland'") ;
```

These queries compare a field called NAME in FATHER.DBF with the string "Ben Nyland". The file name FATHER does not need to be specified in the query because the FATHER.DBF file is a master.

In order to check for the composite record that has Ben Nyland and his son Eric Nyland, the query can be specified by either of the following:

```
relation.querySet( "FATHER->NAME = 'Ben Nyland'.AND.SON->NAME = 'Eric Nyland'") ;
relation.querySet( "NAME = 'Ben Nyland' .AND. SON->NAME = 'Eric Nyland'") ;
```

The query expression can be changed at any time, although **Relate4set::top** or **Relate4set::bottom** must then be called before **Relate4set::skip** can be called.

The Sort Expression

The *sort expression* of a relation is very similar to an index expression. They are both dBASE expressions, and are both used to determine the sort ordering.

The difference is that the sort expression determines the sort order of the query set and you are allowed to use fields from any data file in the relation set.

Setting the sort expression

The sort expression is specified for a relation set by **Relate4set::sortSet**. This function specifies the sort ordering for the entire query set.

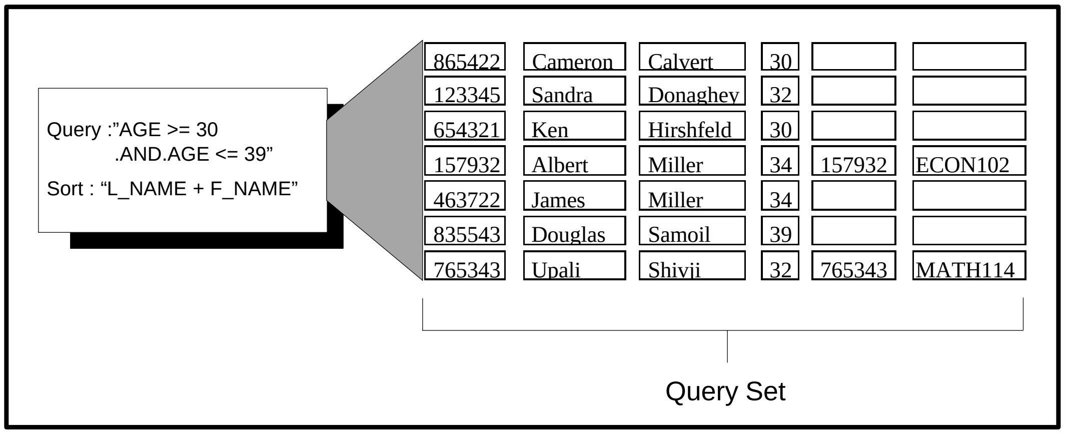
The sort expression can contain field names from any data file in the relation set. It is required that any fields, except those from the top master, are qualified by the data file's alias.

```
relation.sortSet("L_NAME + F_NAME + CLASSES->C_CODE") ;
```

Like the query expression, the sort expression can be changed at any time, although **Relate4set::top** or **Relate4set::bottom** must then be called before **Relate4set::skip** can be called.

Performing Queries on Relation Sets

Queries can be performed on relation sets of any size, including those consisting of a single data file. The following figures illustrate two queries, which are on the composite data file shown in Figure 6.3.



865422	Cameron	Calvert	30		
123345	Sandra	Donaghey	32		
654321	Ken	Hirshfeld	30		
157932	Albert	Miller	34	157932	ECON102
463722	James	Miller	34		
835543	Douglas	Samoil	39		
765343	Upali	Shivji	32	765343	MATH114

Figure 6.8 Query Example One

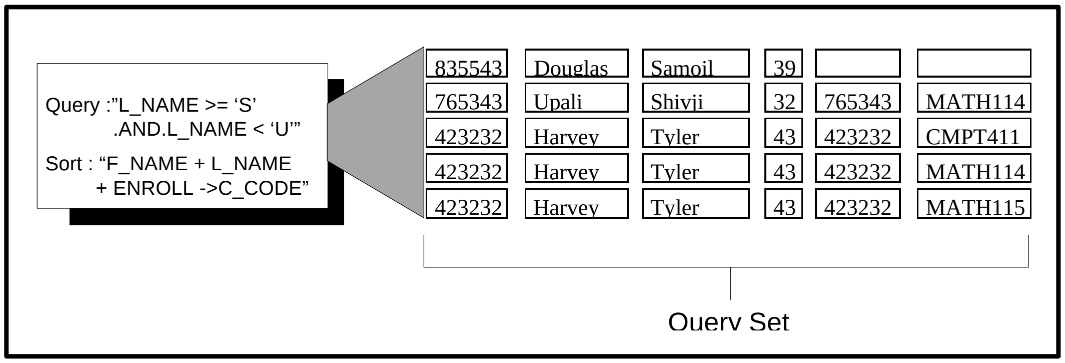

Query :"L_NAME >= 'S'
.AND.L_NAME < 'U'"
Sort : "F_NAME + L_NAME
+ ENROLL ->C_CODE"
835543 Douglas Samoil 39
765343 Upali Shivji 32 765343 MATH114
423232 Harvey Tyler 43 423232 CMPT411
423232 Harvey Tyler 43 423232 MATH114
423232 Harvey Tyler 43 423232 MATH115
Query Set

Figure 6.9 Query Example Two

Accessing The Query Set

The query set is accessed by **Relate4set::top** , **Relate4set::bottom** and **Relate4set::skip**. When a query has been specified, these three functions then access a query set instead of the entire composite data file.

Generating The Query Set

The query set is initially formed by the first call to either **Relate4set::top** or **Relate4set::bottom**. These functions analyze the query expression for the most efficient way of performing the query. They then locate all of records that are in the query set. If a sort expression has been specified, the composite records are sorted. And finally, lookups are performed on the data files in the relation to locate the first or last composite record in the query set.

Note When complex relations and sort expressions are used on a relation set consisting of large data files, **Relate4set::top** and **Relate4set::bottom** can take considerable time to execute.

As long as the query and sort expressions are not changed, further calls to **Relate4set::top** and **Relate4set::bottom** do not cause the query set to be regenerated.

Regenerating the Query Set

You can force CodeBase to completely regenerate the query set by calling **Relate4set::changed**. This forces CodeBase to discard any buffered information and regenerate the query set based on the current state of the relation set's data files. After **Relate4set::changed** is called, **Relate4set::top** or **Relate4set::bottom** must be called before any further calls to **Relate4set::skip** may be made.

Queries On Multi-Layered Relation Sets

The steps involved in performing queries on multi-layered relation sets are the same as on a relation set containing a single data file. You create your relations, set the types of the relations, specify query and sort expressions and retrieve the composite records from the query set.

PROGRAM RELATE3.CPP

Program RELATE3.CPP creates a multi-layered relation between the STUDENT, ENROLL and CLASSES data files and then performs queries on it.

```
#include "d4all.hpp"
extern unsigned _stklen = 10000;

Code4      codeBase ;
Data4      student, enrolment, classes ;
Field4     studentId, firstName, lastName, age, classCode, classTitle ;
Tag4       idTag, codeTag ;
Relate4set classRel ;
Relate4    studentRel, enrollRel ;

void printStudents( void ) ;

void openDataFiles()
{
    student.open( codeBase, "STUDENT" ) ;
    enrolment.open( codeBase, "ENROLL" ) ;
    classes.open( codeBase, "CLASSES" ) ;

    studentId.init( student, "ID" ) ;
    firstName.init( student, "F_NAME" ) ;
    lastName .init( student, "L_NAME" ) ;
    age.init( student, "AGE" ) ;
    classCode.init( classes, "CODE" ) ;
    classTitle.init( classes, "TITLE" ) ;

    idTag.init( student, "ID_TAG" ) ;
    codeTag.init( enrolment, "C_CODE_TAG" ) ;

    codeBase.exitTest( ) ;
}

void printStudents( )
{
    cout << endl << "       " << firstName.str( ) ;
    cout << " " << lastName.str( ) ;
    cout << " " << studentId.str( ) ;
    cout << " " << age.str( ) ;
}

void setRelation( void )
{
    classRel.init( classes ) ;

    enrollRel.init(  classRel, enrolment, "CODE", codeTag ) ;

    studentRel.init( enrollRel, student, "STU_ID_TAG", idTag ) ;
}

void printStudentList( char *expr, long direction )
{
    classRel.querySet( expr ) ;
    classRel.sortSet( "STUDENT->L_NAME + STUDENT->F_NAME" ) ;

    enrollRel.type( relate4scan ) ;

    int rc, endValue ;
    if( direction > 0 )
    {
        rc = classRel.top( ) ;
        endValue = r4eof ;
    }
    else
    {
        rc = classRel.bottom( ) ;
        endValue = r4bof ;
    }
    cout << endl << classCode.str( ) ;
    cout << "  " << classTitle.str( ) ;

    for ( ; rc != endValue ; rc = classRel.skip( direction ) )
        printStudents( ) ;

    cout << endl ;
}

void main( void )
{
    openDataFiles( ) ;

    setRelation( ) ;

    printStudentList( "CODE = 'MATH114 '", 1L ) ;

    printStudentList( "CODE = 'CMPT411 '", -1L ) ;

    codeBase.unlock( ) ;
    classRel.free( ) ;

    codeBase.initUndo( ) ;
}
```

Lookups On Relations

The method illustrated above for retrieving composite records from the composite data file is well suited for performing queries and generating reports, but has unnecessary overhead if all you want to do is perform lookups to slave data files. As a result CodeBase provides an alternative method for performing lookups that is independent of the query set and sort orderings.

PROGRAM RELATE4.CPP

Program RELATE4.CPP sets up an exact match type relation between the STUDENT.DBF and ENROLL.DBF that were illustrated in Figure 6.1. This program performs some seeks and lookups.

```
#include "d4all.hpp"
extern unsigned _stklen = 10000 ;

Code4      codeBase( 0 ) ;
Data4      student, enrolment ;
Field4     id, fName, lName, age, cCode ;
Relate4    slave ; Relate4set master ;
Tag4       idTag, nameTag ;

void openDataFiles( void ) ;
void setRelation( void ) ;
void printRecord( void ) ;

void openDataFiles()
{
    codeBase.init( ) ;

    student.open( codeBa se,"STUDENT" ) ;
    enrolment.open( codeBase,"ENROLL");

    id.init( student,"ID");
    fName.init( student,"F_NAME");
    lName.init( student,"L_NAME");
    age.init( student,"AGE");
    cCode.init( enrolment,"C_CODE_TAG");

    nameTag.init( student,"NAME");
    idTag.init( enrolment,"STU_ID_TAG");

    codeBase.exitTest( ) ;
}

void setRelation()
{
    master.init( student ) ;
    slave.init( master, enrolment, "ID", idTag);
}

void seek( Data4 dataFile, Tag4 tag, Relate4set relation, char *key)
{
    Tag4 oldTag;

    oldTag.initSelected( dataFile ) ;
    dataFile.select( tag ) ;
    dataFile.seek( key ) ;
    relation.doAll( ) ;

    dataFile.select( oldTag ) ;
}

void printRecord()
{
    cout << fName.str( ) << " " ;
    cout << lName.str( ) << " " ;
    cout << id.str( ) << " " ;
    cout << age.str( ) << " " ;
    cout << cCode.str( ) << endl ;
}

void main( void )
{
    openDataFiles( ) ;

    setRelation( ) ;

    seek( student, nameTa g, master, "Tyler           Harvey         ") ;

    printRecord( ) ;

    seek( student, nameTag, master, "Miller          Albert        ");

    printRecord( ) ;

    codeBase.unlock( ) ;
    master.free( ) ;

    codeBase.closeAll( ) ;
    codeBase.initUndo( ) ;
}
```

Performing A Lookup

Performing lookups is quite simple. First you locate the record in the master data file that you wish to perform the lookup from, using the normal **Data4::top**, **Data4::bottom**, **Data4::go**, **Data4::seek, Data4::seekNext** and **Data4::skip** functions. The lookup is then accomplished by a call to **Relate4::doAll**. This function performs lookups on the slave data files of the master data file. The following function performs a seek on a data file and then performs the lookups on it's slaves.

Code fragment from RELATE4.CPP

```
void seek( Data4 dataFile, Tag4 tag, Relate4set relation,
                                                            char *key)
{
    Tag4 oldTag;

    oldTag.initSelected( dataFile ) ;
    dataFile.select( tag ) ;

    dataFile.seek( key ) ;
    relation.doAll( ) ;

    dataFile.select( oldTag ) ;
}
```

Note

Any query and sort expressions are ignored by functions **Relate4set::doAll** and **Relate4::doOne**. Consequently, these functions provide somewhat independent functionality. Using them in conjunction with relate functions such as **Relate4set::top, Relate4set::bottom,** and **Relate4set::skip** is not particularly useful.

Iterating Through The Relations

As an aid in writing generic reusable code, CodeBase provides a class that allows you to iterate through the relations in a relation set. This is the **Relate4iterator** class.

This class, derived from **Relate4**, can be used to move through and access each relation in the relation tree using its two member functions **Relate4iterator::next**, and **Relate4iterator::nextPosition**.

The following code segment displays all of the records from all of the relations in the relation set.

Code Fragment from RELATE1.CPP

```
void printRecord()
{
    Relate4iterator relation( master ) ;
    Data4   data ;
    Field4memo  field;

    for( relation; relation.isValid( ) ; relation.next( ) )
    {
        data = relation.data( ) ;

        for( int j = 1; j <= data.numFields( ); j++ )
        {
            field.init( data, j ) ;
            cout << field.str( ) << " " ;
        }
    }
    cout << endl ;
}
```

7 Query Optimization

What is Query Optimization

Query Optimization is an application invisible method of returning query results at high speed. The CodeBase Relation/Query module (**Relate4** class) uses Query Optimization to analyze the query condition and return the matching set of records with a minimum of disk accesses.

This is accomplished by comparing the query expression to the tag sort orderings of the top master tag. If part of the query matches the tag expression, the tag itself is used to filter out records that do not match the expression.

This results in lightning quick performance, even on the largest data files, since very few records are actually physically read. For example, if there were an index built on PRODUCT and the query was:

PRODUCT = 'GIZMO'

The Query Optimization would automatically use the index file to quickly seek to the appropriate record and, using the PRODUCT tag, and skip to retrieve all the records where PRODUCT is equal to 'GIZMO'. Once this is no longer the case, Query Optimization ignores the remaining records.

It makes little difference whether the data file has 500 records or 500,000 records, since the same number of disk reads are performed in each case.

Complex expressions

Query Optimization can also work on more complex expressions involving the .AND. and .OR. operators. In these cases, Query Optimization breaks down the whole expression into sub-expressions that can be optimized. For example, the two sub-expressions in the query:

L_NAME = "SMITH" .AND. F_NAME = "JOE"

could both be optimized if there were two tags, one based on the expression L_NAME, the other on F_NAME. CodeBase could still partially optimize the query if there was a tag on either of the two sub-expressions. Even a partially optimized query can lead to very fast performance.

When is Query Optimiztion used

To start with, the Query Optimization capabilities are built into the CodeBase relate/query module, the report module and into CodeReporter.   Your applications can use the Query Optimization provided a few simple conditions are met.

Requirements of Query Optimization

To ensure that the Query Optimization is used to return your queries, the following steps must be taken:

1. Insure that the Master data file's index file(s) are open. In general, the more tags open on the master data files, the greater the chance that Query Optimization can take effect.
2. Specify a query expression that contains in whole or in part, a top master tag key compared to a constant.
3. The query expression is set using **Relate4set::querySet**. Query Optimization is only effective when the query expression involves the top master data file.

The following tables illustrate some of the situations where Query Optimization is used automatically, and where it cannot be used. The information in the tables is based on a data file that has the following fields and tags:

```
FIELD4INFO fieldInfo [] =
{
    {"L_NAME", r4str, 10, 0},
    {"F_NAME", r4str, 10, 0},
    {"COMPANY", r4str, 10, 0},
    {"AGE", r4num, 2, 0},
    {"DT", r4date, 8, 0},
    {"AMOUNT", r4num, 7, 2},
    {"ADDRESS", r4str, 15, 0},
    {"COMMENT", r4memo, 10, 0},
    {"PRODUCT", r4str, 10, 0},
    {"ID", r4str, 5, 0},
    {"PHONE", r4str, 8, 0},
    {0,0,0,0},
};

TAG4INFO tagInfo [] =
{
    {"L_NAME_TAG", "L_NAME", 0,0,0},
    {"COMPANY_TAG", "UPPER( COMPANY)", 0,0,0},
    {"AGE_TAG", "AGE", 0,0,0},
    {"DT_TAG", "DT", 0,0,0},
    {"AMNT _TAG", "AMOUNT", ".NOT.DELETED()",0,0},
    {"ADDR_TAG","ADDRESS", "DELETED().AND..NOT.DELETED(), 0,0},
    {"COMMENT_TAG", "COMMENT",  0,0, r4descending},
    {"PROD_TAG", "PRODUCT", 0, r4uniqueContinue, 0},
    {"PRO_ID_TAG", "PRODUCT+ID", 0,0,0},
    {"PHONE_TAG", "PHONE", 0, r4unique, 0},
    {0,0,0,0,0},
};
```

Query Optimization is automatically used in the following situations.

Query Expression	Explanation
L_NAME = "SMITH"	L_NAME is a tag expression which is compared to a constant.
L_NAME >= "S" .AND. L_NAME <= "T"	Each sub expression compares a tag expression against a constant, and so the entire query can use Query Optimization.
L_NAME = "SMITH" .AND. F_NAME = "JOE"	Only the first sub expression can use Query Optimization because there is no tag based on F_NAME.
UPPER(COMPANY) = "IBM"	Again, this is a tag expression compared to a constant. Even though the expression contains a function (UPPER), Query Optimization is still used.
UPPER(COMPANY) = "IBM" .AND. L_NAME="SMITH"	Both parts of the expression use Query Optimization, since both sides compare a tag expression to a constant value.
AGE > 50	Even though the tag is Numeric, the expression can still be optimized.
AGE > 25*2	25*2 is still a constant even though it is a complicated constant.

DTOS(DT) >= "19930101"	This would return all records where the date value in the DT field is greater than or equal to January 1,1993. If the tag expression was DT instead of DTOS(DT), then a query expression such as DT>=CTOD("01/01/93") would also use Query Optimization. (You have to be careful when using CTOD because the format of its parameter depends on the data picture set in **Code4::dateFormat**.)
COMMENT = "SALE"	In this case the tag was set using **r4descending**. Query Optimization can be used because CodeBase is indifferent to the ordering of the tag.
PHONE = "555 6031"	Query Optimization can utilize unique tags as long the tag has not been specified with **r4uniqueContinue. r4unique** prevents duplicate records from being added to the data base. Therefore, **r4unique** does not act as a filter for the tag, unlike **r4uniqueContinue**. Note that when the index is opened, the unique code is automatically set to the default value of **Code4::errDefaultUnique**, which is **r4uniqueContinue.** Use **Tag4::unique** to set unique code to **r4unique** before making a query. See the "Unique Keys" section of the "Indexing" chapter of this guide for more information.

The following expressions cannot use the Query Optimization. Queries on these expressions are not optimized and take a longer time to execute.

Query Expression	Explanation
F_NAME = "JOE"	There is no tag based on F_NAME.
COMPANY = "IBM"	There is no tag based on COMPANY. The tag is UPPER(COMPANY). Again, the tag expression must match the expression being compared to the constant exactly.
L_NAME = F_NAME	This expression cannot be optimized because F_NAME is not a constant.
AMOUNT = 1000.00	Query Optimization will not be used because the tag contains a filter.
ADDRESS = "104 ELM ST"	Query Optimization will not be used because the tag contains a filter, even though the filter is useless. Tags with filters are useless, regardless of the filter result.
PRODUCT ="GIZMO"	Query Optimization will not be used because the tag specifies **r4uniqueContinue**. **r4uniqueContinue** acts like a filter, allowing duplicate records in the database but not in the index.
PRODUCT + ID ="GIZMO 13445"	Query Optimization will not be used because the tag consists of two fields.

How To Use Query Optimization

Query Optimization is automatically enabled whenever a relation operation occurs. CodeBase uses Query Optimization on the top master data file transparently when retrieving records from the composite data file.

Query Optimization, as described above, operates only on the top master data file and only when a tag sort ordering corresponds to a portion of the query expression. All you need to do is ensure that the top master data file has an appropriate index file open.

How to tell if Query Optimization can be used

It is possible to determine whether Query Optimization can be used by calling the function **Relate4set::optimizeable.** This function returns true (non-zero) when the relation is able to use the Query Optimization and it returns false (zero) when it can not. If there is insufficient memory, the Query Optimization will not be invoked, even if **Relate4set::optimizeable** returns true (non-zero).

Call **Relate4set::optimizeable** before functions that may use Query Optimization. If false (zero) is returned, it may be possible to create a new index file for those queries that can not use Query Optimization. In the above scenario, the expression "F_NAME = 'JOE' ", is not able to use Query Optimization. If a new index were built on "F_NAME", the execution time of the query would be improved.

Memory Requirements of Query Optimization

The activation of the Query Optimization also depends on the compiler. Therefore the following restrictions apply:

When a 16 bit compiler is used, the Query Optimization will be able to handle 500,000 records. If the data base has more than half a million records, the query will be made without using Query Optimization.

It is recommended that 32 bit compiler be used when using data bases that have more than 500,000 records. Query Optimization will be able to handle 32 billion records when a 32 bit compiler is used.

8 Date4 Class Functions

The date functions allow you to convert between a variety of date formats and to perform date arithmetic. The date functions also allow you to retrieve the various components of a date, such as the day of the week, in numeric form.

Date Pictures

The format of a date string is represented by a date picture string. A date picture is a string containing several special formatting characters and other characters. The special formatting characters are:

- **C**   Century. A 'C' represents the first digit of the century. If two 'C's appear together, then both digits of the century are represented. Additional 'C' s are not used as formatting characters.
- **Y**   Year. A 'Y' represents the first digit of the year. If two 'Y's appear together, they represent both digits of the year. Additional 'Y' s are not used as formatting characters.
- **M**   Month. One or two 'M's represent the first or both digits of the month. If there are more than two consecutive 'M's, a character representation of the month is returned.
- **D**   Day. One or two 'D' s represent the first or both digits of the day of the month. Additional 'D' s are not used as formatting characters.
- **Other Characters**   Any character which is not mentioned above is placed in the date string when it is formatted.

For example, if the date August 4 1994 is converted to a date string using the date picture "MM/DD/CCYY", the resulting date string is "08/04/1994". If the same date is converted with the date picture "MMMMMMMM DD, YY" the resulting date string is "August 04, 94".

Date Formats

Dates can be stored in many formats. The two used by CodeBase are the standard format and the Julian day format.

PROGRAM DATE1.CPP

Program DATE1.CPP uses several of the date functions to calculate the number of days until Christmas and until your next birthday.

```
#include "d4all.hpp"

int  validDate( Date4 &date )
{
    if( long( date ) < 1)
        return 0;
    else
        return 1;
}

void howLongUntil( Date4 &tillDate, char *title )
{
    Date4 today ;
    today.today( ) ;

    cout << "Today's date is " << today.format( "MMM DD/CCYY" ) << endl ;

    int thisYear = today.year( ) ;

    Str4len year( tillDate.ptr( ), 4 ) ;
    year.assignLong( today.year( ) ) ;

    if( tillDate < today )
    {
        thisYear ++;
        year.assignLong( thisYear ) ;
    }

    long days = long(tillDate) - long(today) ;
    cout << "There are " << days << " days until "
        <<  title << endl
        << "(which is a " << tillDate.cdow( )
        << " this year)" << endl << endl ;
}

void main( void )
{
    Date4 christmas( "19801225" ) ;
    howLongUntil( christmas, "Christmas");

    Date4 birthday ;

    do
    {
        char birthdate[ 80 ] ;
        cout << "Please enter your birthdate in "
            << "\"DEC20/1993\" format: " ;
        cin >> birthdate ;

        birthday.assign( birthdate, "MMMDD/CCYY" ) ;
    } while( !validDate( birthday ) ) ;

    howLongUntil( birthday, "your next birthday");
}
```

Standard Format

Dates are stored in the data file and internally in the **Date4** class in "CCYYMMDD" format. This is known as the standard format. **Str4::str** (when called from a **Field4** object) returns dates from a Date field in this standard format. **Str4::assign** performs the opposite operation by storing a standard format date string in the data file.

Note If **Str4::assign** from a **Field4** object is passed a date string in anything other than standard format, indexing and seeking will not be performed correctly. It is recommended that **Date4** objects be used when doing assignments to date fields.

Julian Day Format

Another important date format is the Julian day format. A Julian day is defined as the number of days since Jan. 1, 4713 BC. Having the date accessible in this format lets you perform date arithmetic. Two dates can be subtracted to find the number of days separating them, or an integer number of days can be added to a date. The **Date4** class overloads **Str4::operator long** to return Julian dates.

Manipulating dates

Performing manual calculations and conversion between Julian days and the dBASE standard date format is facilitated by using the **Date4** class and its operators.

Adding to a date

All date arithmetic is done in Julian days. A date object (which stores the date in standard format) is converted to a **(long)** Julian date and a number of days are added or subtracted. Once the proper date is obtained, the long value may either be used directly, or stored back into the **Date4** object.

CodeBase provides a number of operators which make this process extremely convenient. In general, **Date4** objects may be treated as if they were **(long)** values. To add one day to a **Date4** object, simply use the increment operator '++' or the increment and assign operator '+='.

```
Date4 today ;
today.today( ) ;

today++ ; // 'today' now contains tomorrow.
today+=6L ; // 'today' now contains a week from now
```

Following the same concept, a date may be added to without modifying the **Date4** object by using the binary addition operator '+'. This operator returns a **(long)** Julian date value for the sum of the date and the number to be added.

```
Date4 tomorrow, today ;
today.today( ) ;
tomorrow = today + 1L ;

if( today + 1L == long(tomorrow) )
    cout << "This is always true" ;
```

Subtraction of days from a **Date4** object is accomplished using the subtraction operators: **--**, **-=**, and **-**. These operators perform in the same manner as the addition operators.

Comparing Dates

Date comparisons are accomplished using the standard C++ relational operators: ==, !=, >, >=, <, <=. As the following code fragment illustrates, the comparison is straight forward.

Code fragment from DATE1.CPP

```
if( tillDate < today )
{
    thisYear ++;
    year.assignLong( thisYear ) ;
}
```

Comparing **Str4** objects to Dates

Since the **Date4** class is derived from **Str4**, any **Str4**-derived class object may be compared to a date. For equality to be returned from the ==, !=, >=, and/or <= operators, the **Str4** object must store a date value in standard character format, and have a length of 8.

```
Date4 date( "19801112" ) ; // November 12, 1980
Str4ten string ;
string.assign( "19801112" ) ;

if( string == date )
    cout << "This is a lways true" << endl ;
```

Since data base **Field4** objects are derived from **Str4**, these comparison operators may be directly used with date fields. Below is a code sample which demonstrates the comparison of **Date4** objects with a date field.

```
Field4 dateField( data, "BIRTH_DATE" ) ;
Date4 dateObject( "19801112" ) ; // Nov. 12, 1980

for( data.top( ); !data.eof( ); data.skip( ) )
    if( dateField > dateObject )
    {
        cout << endl << "Record #" << data.recNo( )
            << " contains a d ate greater than "
            << dateObject.format( "MMM. DD, CCYY" ) ;
    break ;
    }
```

Comparing Julian dates to dates

The easiest, and most efficient way to compare a Julian date value to a **Date4** object is to use the **Date4::operator long** operator to convert the **Date4** object to a long value. The standard C++ operator is then used to compare the long values.

Another method is to construct a temporary **Date4** object around the **(long)** value. Both methods are shown below:

```
Date4 date( "19801112" ) ;
long l = long( date ) ; // store the Julian date

cout << "(long) value of the date is " << l << endl ;

if( l == long( date ) )
    cout << "This will always happen" << endl ;

if( Date4( l ) > date )
    cout << "This will never happen" << endl ;
    // uses Str4::operator > to compare objects.
```

Assigning dates to fields

Date4 objects can be assigned to data base date fields in the exact same manner as any other **Str4** derived object: with **Str4::assign**.

Field4 objects, being derived from **Str4**, use the **Str4::assign** to store information in the data file. One version of **Str4::assign** may take a **Str4** object as a parameter. It is this version which is used with **Date4** objects (derived from **Str4**) to safely assign dates to data base Date fields.

```
Field4 dateField( data, "BIRTH_DATE" ) ;
Date4 dateObject( "19801112" ) ;

data.top( ) ;
// store Nov. 12, 1980 in the birth date of the first record.
dateField.assign( dateObject ) ;
```

Testing For Invalid Dates

You can test for invalid dates using the **Date4::operator long** operator. If the date object contains an invalid date, this operator will return **'(long) -1'**. If the **Date4** object is uninitialized, or is assigned a blank string, the operator returns zero. The following example function tests for valid dates:

Code fragment from DATE1.CPP

```
int  validDate( Date4 &date )
{
    if( long( date ) < 1)
        return 0;
    else
        return 1;
}
```

Other Date4 Functions

There are several other **Date4** member functions that perform various date related tasks:

Days and Months as Characters

There are two functions that return a portion of the **Date4** object as a character strings. They are:

Date4::cdow The day of the week in character form is returned.

Date4::cmonth The day of the month in character form is returned.

Code fragment from DATE1.CPP

```
cout <<  title << endl
    << "(which is a " << tillDate.cdow( )
    << " this year)" << endl << endl ;
```

Days, Months and Years as Integers

There are four functions that return a portion of the **Date4** object as integer values:

- **Date4::day** The day of the month is returned as an integer value from 1 to 31.
- **Date4::dow** The day of the week is returned as an integer value from 1 to 7.
- **Date4::month** The month of the year is returned as an integer value from 1 to 12.
- **Date4::year** Returns the century/year portion of the date as an integer value.

```
void calcMonthStats( Data4 data, long *month )
{
    memset( months, 0, sizeof( long )* 13 ) ; // 12 months

    Field4 transactionDate( data, "TRANSDATE" ) ;
    Date4 date ;

    for( int rc = data.top( ); rc == r4success; rc   = data.skip( ) )
    {
        date.assign( transactionDate ) ;
        months[ date.month( ) ] ++ ;
    }
}
```

9 Linked Lists

The C++ language has excellent support for arrays. You can create arrays of predefined types such as integers or doubles, or you can create arrays of your own types and structures. Although arrays are highly useful constructs, they have some major limitations: you cannot insert new elements between existing elements, you cannot remove existing elements, and you have to allocate all of the array's memory at once (either at compile time or run time). Although there are ways around the limitations of arrays, these methods are generally inefficient and involve shuffling large amounts of memory.

The Linked List

A more elegant way of providing this functionality is to use linked lists. A linked list is an group of connected elements called *nodes*. Each node contains data and pointers to other nodes.

The list can be constructed in the following manner: the pointer of the first node points to the second node, and the pointer of the second node point to the third node and so on. The pointer of the last node contains null (in this manual, a pointer which contains null is called a null pointer).

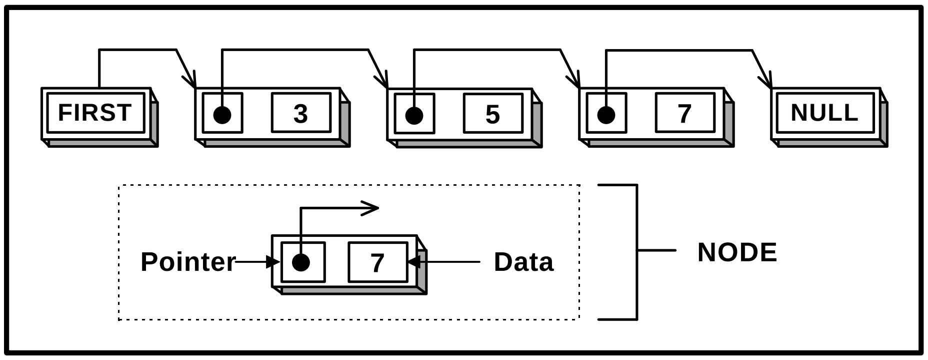


Figure 10.1 A Linked List

The advantages of linked lists over arrays are that the nodes can be located anywhere in memory, new nodes can be added at any time, and nodes can be added and deleted from the list by simply changing the value of a few pointers.

Double Linked Lists

The linked lists used by CodeBase are *double linked lists*. A double linked list is similar to the type of linked list described above except that each node has two pointers instead of one. The first pointer points to the next node, while the second points to the previous node. The following is a standard double linked list.

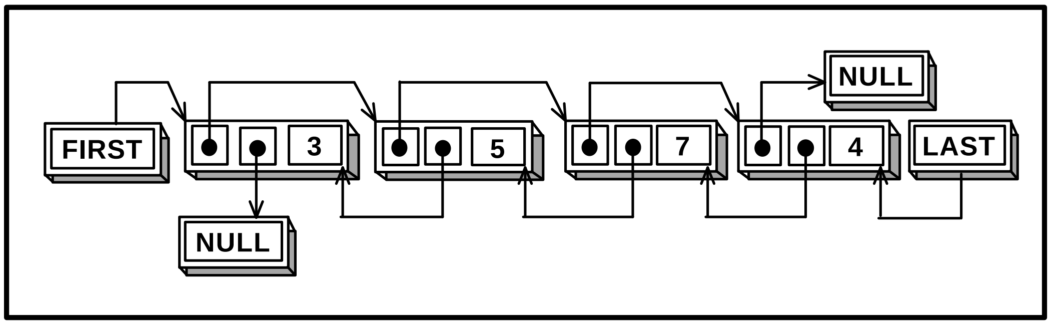


Figure 10.2 Double Linked List

This allows you to traverse the linked list from both directions starting from pointers to the first and last nodes. In addition, it allows you to remove a node without traversing the list.

CodeBase Linked Lists

To implement a linked list using CodeBase, there are two requirements. The first requirment is a **List4** object and the second is a user defined structure or class for the nodes.

PROGRAM LIST1.CPP

The program "LIST1.CPP" demonstrates how to traverse, add and remove nodes from a linked list.

```
#include "d4all.hpp"
extern unsigned _stklen = 10000 ; // for all Borland compilers

class Ages : public LINK4
{
    public:
        Ages( int newAge ) { age = newAge ;}
        int age;
} ;

void printList( List4 &ageList )
{
    Ages    *nodePtr;

    cout << endl << "There are " << ageList.numNodes( )
        << " links." << endl ;

    nodePtr = (Ages *) ageList.first( ) ;
    while( nodePtr != NULL )
    {
        cout << nodePtr->age ;
        nodePtr = (Ages *) ageList.next( nodePtr ) ;
    }
}

void main( void )
{
    List4   ageList ;
    Ages    firstAge( 3 ), middleAge( 5 ), lastAge( 7 );

    ageList.add( &middleAge ) ;
    ageList.addBefore( &middleAge, &firstAge ) ;
    ageList.addAfter( &middleAge, &lastAge ) ;

    printList( ageList ) ;

    ageList.remove((void *) &middleAge);

     printList(  ageList ) ;

    ageList.pop( ) ;

    printList( ageList ) ;
}
```

The List4 Class

The **List4** class contains the control information for the linked list. It has a counter containing the number of nodes in the linked list and a pointer to the selected node.

Constructing a **List4** object

Each linked list that your application contains will have its own **List4** object. Since the linked list functions are independent of the CodeBase data base management, a **Code4** object does not need to be constructed.

Code Segment From "LIST1.CPP"

```
 List4 ageList ;
```

The Nodes

The node of a CodeBase linked list is a data structure or class that you define. This may be any class or structure, as long as the first member is a **LINK4** structure.

Example Node Class

```
class Ages : public LINK4
{
    int age ;
} ;
```

Example Node Structure

```
typedef struct
{
LINK4   link;
int     age;
} AGES ;
```

The **LINK4** Structure

The **LINK4** structure contains pointers to the next and previous nodes in the list. These pointers are maintained by CodeBase functions so there is no need for you to access them.

Adding Nodes to Linked Lists

There are three functions that add new nodes to CodeBase linked lists. They are **List4::add**, **List4::addBefore** and **List4::addAfter**.

The **List4::add** functions adds the new node to the end of the list. This function requires only one parameter: a pointer to the node to be added.

Note

Pointers to the node are passed to CodeBase linked list functions as void pointers. Any time you see parameters that are of type (void *), you can assume that CodeBase expects a pointer to a node.

Here is a code fragment that assigns some values to the example node structures and adds one of them to the linked list.

Code Segment using structures

```
    AGES    firstAge, middleAge, lastAge;
    .
    .
    .
    /* assign values to the nodes */

    firstAge.age = 3;
    middleAge.age = 5;
    lastAge.age = 7;

    /* add a node to the linked list */
    ageList.add( &middleAge ) ;
```

Inserting Nodes

List4::addBefore inserts a node into the linked list directly before the given node. **List4::addAfter** inserts a node directly after the given node.

In the example program LIST1.CPP, the new nodes are added before and after a node in the linked list.

Code Segment From "LIST1.CPP"

```
    ageList.addBefore( &middleAge, &firstAge) ;
    ageList.addAfter( &middleAge, &lastAge ) ;
```

At this point, the *ageList* linked list would look as follows:

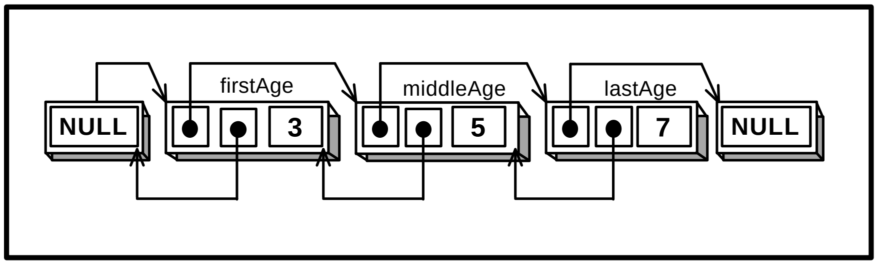


Figure 10.3 Linked List after several additions

Traversing The Linked List

Once items have been added to the linked list, you can then traverse the list from either direction.

Retrieving the First and Last Nodes

The **List4** class lets you retrieve the first and last nodes in the list by calling **List4::first** and **List4::last**, respectively. Both of these functions will return a null pointer if there are no nodes in the linked list.

Obtaining the Next Node

List4::next returns a pointer to the next node in the linked list. For example if you passed **List4::next** a pointer to *middleAge*, it would return a pointer to *lastAge*. You can also use **List4::next** to obtain a pointer to the first node by passing it a null pointer. You can combine these functions to iterate through the list. This is illustrated by the following code fragment that prints out all the nodes in a linked list:

Code Segment From "LIST1.CPP"

```
nodePtr = (Ages *) ageList.first( ) ;
while( nodePtr != NULL )
{
    cout << nod ePtr->age ;
    nodePtr = (Ages *) ageList.next( nodePtr ) ;
}
```

The first node is located by **List4::first**. To avoid compiler errors, the return value from **List4::first** is cast as **(Ages *).**

Note

List4::first, **List4::last**, **List4::next**, **List4::prev**, and **List4::pop** all return **(void *)** pointers that actually point to node. To avoid compiler errors you must cast the return values of these functions as node pointers.

Next, a **while** loop is entered and the contents of the node are printed out. Before the loop terminates, the next node of the linked list is located by a call to **List4::next**. After this call, *agePtr* is either a pointer to the next node, or null if there are no more items in the list.

Traversing The List in Reverse Order

If you want to move through the list from the end to the beginning, you would use **List4::last** and **List4::prev**. **List4::prev** behaves the same as **List4::next** except it returns a pointer to the node before the specified node.

The Node Count

If you need to know how many nodes there are in a linked list, **List4::numNodes** may be used. This returns the number of nodes currently in the list. Since this value is automatically updated each time the list is modified, it is always current.

Removing Items From A Linked List

There are two functions that remove items from a linked list. The first is **List4::pop**. This function removes the last node from the linked list. **List4::pop** returns a pointer to the node being removed, so that you can free the memory associated with the node. If you want to remove a specific node from anywhere in the linked list, **List4::remove** is used.

Dynamic Allocation

The most powerful feature of linked lists is the ability to dynamically allocate nodes (i.e. allocate nodes at run-time). This section will show how you can dynamically add and remove nodes from linked lists.

PROGRAM LIST2.CPP

The "LIST2.CPP" is a modification of "LIST1.CPP". In this program the nodes are dynamically allocated and freed.

```
#include "d4all.hpp"
extern unsigned _stklen = 10000 ; // for all Borland compilers
class Ages : public LINK4
{
public:
    Ages( int newAge ) { age = newAge ;}
    int age;
} ;

class AgeList : public List4
{
public:
    ~AgeList( ) { freeAges( ) ; }
    Ages *addAge( int ) ;
    void freeAges( ) ;
    void removeAge( Ages * ) ;
    void printList( ) ;
} ;

Ages *AgeList::addAge( int nAge )
{
    return (Ages *) add( new Ages ( nAge ) );
}

void AgeList::removeAge( Ages *rAge )
{
    remove( rAge );
    delete rAge;
}

void AgeList::freeAges( )
{
    Ages *a ;
    while( (a = (Ages *) pop( )) != NULL )
        delete a ;
}

void AgeList::printList( )
{
    Ages    *nodePtr;
    cout << endl << "There are " << numNodes( ) << " links." << endl ;

    nodePtr = (Ages *) first( ) ;
    while(nodePtr != NULL)
    {
        cout << nodePtr->age ;
        nodePtr = (Ages *) next( nodePtr ) ;
    }
    if( numNodes( ) )
        cout << endl << ((Ages *) selected)->age << " is selected " << endl ;
}
```

```
void main( void )
{
    AgeList   ageList ;

    ageList.addAge( 3 ) ;
    Ages *agePtr = ageList.addAge( 5 ) ;
    ageList.addAge( 7 ) ;
    ageList.addAge( 6 ) ;
    ageList.addAge( 2 ) ;

    ageList.selected = agePtr ;

    ageList.printList( ) ;

    ageList.removeAge( agePtr ) ;
    ageList.printList( ) ;

    ageList.freeAges( ) ;
    ageList.printList( ) ;
}
```

Adding Nodes

The process of dynamically adding a node to a linked list involves two steps. First you must allocate the memory for the node and then add that node to the linked list using a **List4** member function.

Allocating Memory

Allocating memory can be performed using the C++ operator **new**. This operator allocates global memory for the object and calls its constructor. Since this operator is part of the C++ language, it is extremely portable between all operating systems. Alternatively, the CodeBase utility function **u4alloc** may be used to perform portable memory allocation.

AgeList::addAge in "LIST2.CPP" uses **new** to dynamically allocate a node and add it to the linked list.

Code Segment From "LIST2.CPP"

```
Ages *AgeList::addAge( int nAge )
{
    return (Ages *) add( new Ages( nAge ) );
}
```

Removing Nodes

When you dynamically allocate nodes in a linked list, it is important that you deallocate them when you are finished with them. If you do not, the memory is not returned to the system until your application terminates.

Freeing Memory

Deallocation is performed with the C++ **delete** operator. You provide it a pointer to memory that was previously allocated with the **new** operator, and it is freed and returned to the system. If the memory was allocated with CodeBase function **u4alloc**, **u4free** should be called to free the memory to the system.

Code Segment From "LIST2.CPP"

In "LIST2.CPP" the *AgeList::removeAge* member function removes a node from the linked list and then frees its memory.

```
void AgeList::removeAge( Ages *rAge )
{
    remove( rAge );
    delete rAge;
}
```

Cleaning Up The List

It is always good programming practice to free up any memory that is allocated by your application. When you are finished using your linked list, you should deallocate all of its nodes. This is usually done in a destructor for a class. When the linked list object falls out of scope, the destructor is called and all the nodes are freed.

Using **List4::pop** in a **while** loop is an efficient technique of freeing the nodes.

AgeList::freeAges, in "LIST2.CPP" removes and deallocates all of the nodes from the linked list. Notice also that the AgeList destructor also calls *AgeList::freeAges*.

Code Segment From "LIST2.CPP"

```
void AgeList::freeAges( )
{
    Ages *a ;
    while( (a = (Ages *) pop( )) !=  NULL
        delete a ;
}
```

Stacks And Queues

In addition to the standard type of linked list functions, you can also use the linked list functions to implement *stacks* and *queues*.

Stacks

A stack is a list of nodes that can only be added or removed from one end. A deck of playing cards can be an example of a stack. Cards are added by placing them on top of the deck (traditionally called *pushing* a node onto the stack), and removed by taking the top card from the deck (traditionally called *popping* an node from the stack). CodeBase provides two functions, **List4::add** and **List4::pop,** that provide these behaviours.

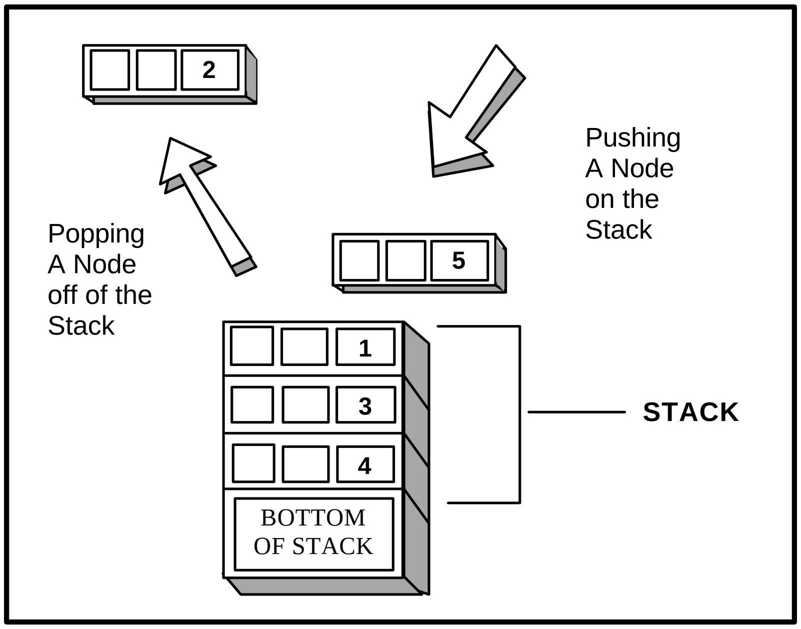


Figure 10.4 Popping and Pushing node on a Stack

Queues

A queue is similar to stack except that nodes are added at one end and removed from the other. A line at the movie box office is an example of a queue. People line up at the back and leave from the front once they have purchased their tickets.

Queues can also be easily implemented using CodeBase linked lists. To remove a node from the queue, **List4::pop** is used. To add a node to the list **List4::first** and **List4::addBefore** are used.

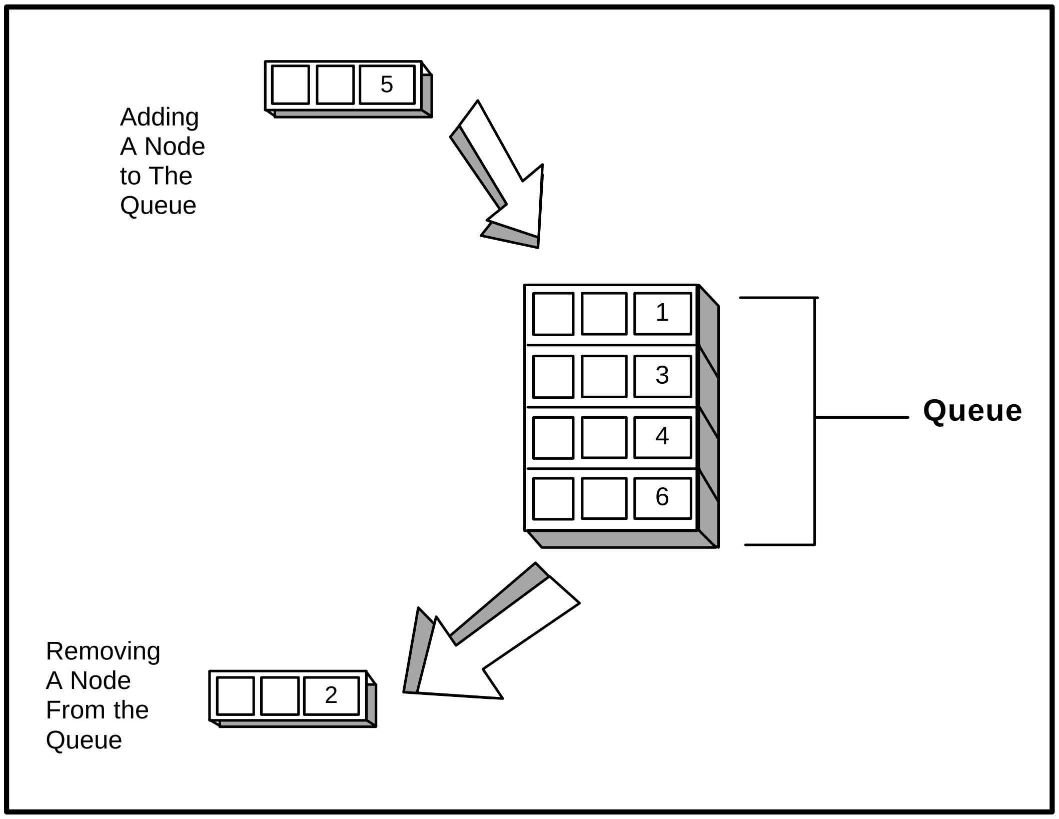


Figure 10.5 Adding and Removing Items from a Queue.

10 Memory Optimizations

Memory optimization is accomplished by buffering portions of data, index and memo files in memory.   This improves performance because physically accessing the disk takes longer than accessing memory. In addition, reading and writing several records at once is considerably quicker than reading and writing individual records separately.

Memory optimization is more effective in some operating environments than others. This is because some operating systems already do some memory optimizations at the operating system level and some hard disks do memory optimizations at the hardware level.

Regardless, when memory is available, proper use of CodeBase memory optimizations will always improve performance. This is because CodeBase memory optimizations are designed specifically to optimize database operations. In addition, in network applications, they can reduce requests to the network server.

Using Memory Optimizations

Using memory optimization is quite easy. All you have to do is turn it on. CodeBase has defaults for determining which files should be optimized and whether the files should be optimized for reading and/or writing. If you wish, you can override the defaults for any particular file(s).

Specifying Files To Optimize

Any file that you can open with a CodeBase function can be optimized. This includes data files, index files and memo files, along with files that are opened with **File4::open**. Each file opened by CodeBase has two flags associated with it. The first specifies whether the file is optimized. When a file is optimized, the second flag specifies whether the file is optimized for both reading and writing or just reading.

There are two ways of setting a file's optimization flags. You can use the **Code4** default settings when the file is opened or you can explicitly set them.

Changing The Default Settings

When the file is opened, its optimization flags are set from the **Code4::optimize** and **Code4::optimizeWrite** flag settings. Valid settings for **Code4::optimize** are:

- **OPT4EXCLUSIVE** (Default) Read optimize files when the files are opened exclusively, when the **S4OFF_MULTI** compilation switch is defined or when the DOS read-only attribute is set. Otherwise do not read-optimize the file.
- **OPT4OFF** Do not read optimize the file.
- **OPT4ALL** Same as **OPT4EXCLUSIVE** except that shared files are also optimized.

Valid settings for **Code4::optimizeWrite** are:

- **OPT4EXCLUSIVE** (Default) Write optimize files when they are opened exclusively or when the **S4OFF_MULTI** compilation switch is defined. Otherwise do not write optimize.
- **OPT4OFF** Do not write optimize.
- **OPT4ALL** Same as **OPT4EXCLUSIVE** except that shared files which are locked are also write optimized.

Note If read optimization is disabled, then write optimization is also disabled.

The following code segment opens three data files and changes their optimization flag settings. When memory optimization is activated, the first data file is read optimized, the second is read/write optimized, and the third has no optimization at all.

```
Code4  codeBase;
Data4  d1, d2, d3;

codeBase.optimize = OPT4ALL ;
codeBase.optimizeWrite = OPT4OFF ;
d1.open( codeBase, "DATA1" ) ;

codeBase.optimizeWrite = OPT4ALL ;
d2.open( codeBase, "DATA2" ) ;

codeBase.optimize = OPT4OFF ;
codeBase.optimizeWrite = OPT4OFF ;
d3.open( codeBase, "DATA3" ) ;
```

Overriding The Default Settings

A file's optimization flags can be changed at any time by calling one of the following functions: **Data4::optimize, File4::optimize, Data4::optimizeWrite,** or **File4::optimizeWrite**. Both the **File4::optimize** and **File4::optimizeWrite** functions change the optimization flags of a single file while the **Data4::optimize** and **Data4::optimizeWrite** change the optimization flags of a data file and all of its open index and memo files. Following is the above example converted to use the **Data4::optimize** and **Data4::optimizeWrite** functions.

```
Code4  codeBase ;
Data4  d1, d2, d3;

d1.open( codeBase, "DATA1" ) ;
d1.optimize( OPT4ALL ) ;
d1.optimizeWrite( OPT4ALL) ;

d2.open( codeBase, "DATA2" ) ;
d2.optimize( OPT4ALL ) ;
d2.optimizeWrite( OPT4ALL ) ;

d3.open( codeBase, "DATA3" ) ;
d3.optimize( OPT4OFF ) ;
d3.optimizeWrite( OPT4OFF ) ;
```

Activating The Optimizations

When the **Code4::optStart** function is called, the files whose optimization flags are set are memory optimized. Memory optimization is disabled by **Code4::optSuspend**.

```
codeBase.optStart( ) ;
// . . .
codeBase.optSuspend( ) ;
```

Refreshing The Buffers

You can also force a refresh of the optimization buffers. **File4::refresh** causes any buffered portion of a file to be discarded. The **Data4::refresh** performs the same function for a data file and its index and memo files. Finally, the **Data4::refreshRecord** rereads the current record from the disk.

Memory Requirements

When using memory optimization, you can limit the amount of memory which CodeBase uses for this purpose by setting **Code4::memStartMax**. The natural question is what is an appropriate maximum? In general, the best maximum is the amount of memory which is likely to be available. For more information please refer to the "**Code4"** class chapter of the *Reference Guide*.

Since most applications run under a number of hardware environments where varying amounts of memory are available, the application should assume that all extra available memory was used by CodeBase memory optimization. Therefore, the application should allocate its own memory before calling **Code4::optStart**, or after calling **Code4::optSuspend**. If you must allocate memory when the optimization is active, use **u4allocFree**. If this function fails to allocate memory, it will free memory from the CodeBase memory buffers and try again. These techniques make CodeBase memory optimization a lower priority.

If you expect very little memory to be available for CodeBase memory optimization, you should probably just not use it. In this case, you might want to build the CodeBase library using the conditional compilation switch **S4OFF_OPTIMIZE** in order to reduce executable size.

In the client-server configuaration, the **S4CLIENT** switch automatically defines **S4OFF_OPTIMIZE** as the default setting.

When To Use Memory Optimization

Using memory optimization is not especially difficult. The most difficult part about memory optimization is knowing when to use it.

Single User

The single user case is the most straight forward. Essentially, a single user application can safely memory read-optimize all files. Write optimization can safely be used. If an application explicitly flushes to disk by calling CodeBase flushing functions, write optimizations are ineffective. In all other single user cases write optimization is useful for improving performance.

When writing single user applications with memory optimization, it is a good idea to set **Code4::accessMode** to **OPEN4DENY_RW**. This way CodeBase performs memory optimizations by default.

Multi-User

Whether or not to use memory optimization in a multi-user or multi-tasking environment is a more difficult decision. This is because memory optimization interferes with the multi-user sharing of information. On the other hand, the speed improvements resulting from the use of memory optimization can be even more dramatic because accessing a network server can be slower than accessing a local drive. In addition, memory optimization causes the server to be used less which can improve response time for other users.

Using memory read optimization in a network environment means that previously read database information is buffered in memory. The next time the information needs to be read, it can be quickly fetched from local memory. The disadvantage of this is that if the information has been changed by another application, the most recent piece of disk information may not be retrieved. At worst, one half of a read record could be an up-to-date version and the other half an older version. If read optimization is being used on memo files being updated by another user, it is theoretically possible to be returned an old or garbled memo entry. If the file is locked, using read optimization is completely safe because it cannot be updated by another user. Consequently, it is necessary to be careful to only use read optimization under appropriate circumstances.

Using memory write-optimization on shared files is potentially even more hazardous than using read-optimization. This is because information is written to disk only when the memory buffers are full. Consequently, from the perspective of any user reading the changed information, the information can appear as corrupt. If other applications are using index files which appear corrupt, they can generate errors. It is not quite as bad as it might sound because CodeBase is programmed with extensive error checking and reacts appropriately to most apparent corruption.

WARNING

Allowing one application to write optimize an index file, while another application may use the same index file, can lead to unpredictable results in the application reading the index file.

Note

CodeBase only allows write optimization when the entire data file is locked or when it is opened exclusively. This restriction is necessary in order to guarantee that index and memo files are not corrupt after they have been flushed.

For more examples of memory optimization, refer to the "Optimization" section of the "Multi-user Applications" chapter.

11 Multi-User Applications

A multi-user application can take many forms. For example, several people could be entering data, over a local area network, into the same data file at the same time. Another possibility is one person entering data while several others look at the data file - all using terminals attached to a powerful computer running UNIX.

A third case is a single user accessing data in a multi-tasking or multi-threading environment, such as Microsoft Windows, OS/2, or UNIX.

When using CodeBase, you have many options in how you design the multi-user aspects of your application. For example, you can lock data areas before you read them or you can read them without locking. You can choose memory optimizations to improve performance or you can write/read directly to/from disk in order to ensure information is current. The exact options you choose depend on the requirements of the application and hardware resources available.

Locking

At the heart of multi-user applications is locking. Locking is a way in which multi-user database applications communicate with each other. When an application locks some data, it is telling other applications "you cannot modify this data". When data is locked, other applications can still read the data. However, no other application can lock or modify the data.

Operations that require locking

Locking is automatically performed by CodeBase before data is written to disk. It is necessary to lock data before it is written to keep two applications from updating the same data at the same time. This avoids the corruption due to several applications updating the same index or memo file at the same time.

When a field is to be modified, the record should be locked to prevent multiple users from changing the same record at the same time. This principle is enforced when **Code4::lockEnforce** is true (non-zero) and the field is modified with a field function or the following **Data4** functions: **Data4::blank**, **Data4::changed**, **Data4::delete** or **Data4::recall**.

Sometimes it is appropriate for an application to lock data before reading it. This is done to keep other applications from updating the data while the lock is present.

Supported Locking Protocols

CodeBase supports a variety of locking protocols that allow applications to be multi-user compatible with applications built using other products. When you build a CodeBase library that uses a specific type of index file compatibility, you also automatically get the multi-user compatibility. The supported locking protocols are as follows:

- **FoxPro** (This is the default locking protocol). This locking protocol is used when the CodeBase library is built with the **S4FOX** conditional compile switch. This provides multi-user compatibility with FoxPro.
- **dBASE IV**   An application is multi-user compatible with dBASE IV when it is built using the **S4MDX** conditional compile switch.
- **Clipper**   Clipper multi-user compatibility is provided by building the CodeBase library with the **S4CLIPPER** conditional compile switch.

For more information on conditional compile switches please refer to the *Getting Started* booklet or the "Conditional Compilational Switches" chapter of the *Reference Guide*.

Types Of Locking

CodeBase performs several types of locking, although the only locks that you need to be concerned with are record and file locks. The types of locks are listed as follows:

- **Record Locking** When a record is locked, that data file record cannot be updated by other applications. This is the lowest level of locking (ie. you cannot lock fields).
- **Data File Locking** When a data file is locked, no records in the data file may be updated by other applications. In addition, a data file lock means that no other application may append records while the data file lock is in place.
- **Index and Memo Locking** CodeBase often locks and unlocks index and memo files when they are updated. However, you do not need to be concerned about this since it is automatic.

Creating Multi-User Applications

Creating well behaved multi-user applications, that is applications that do not lock portions of files for long lengths of time, is relatively easy using CodeBase. If you use the default **Code4** flag settings, there are only a few things you must consider when writing multi-user applications.

Recommended Code4 Flag Settings

The following **Code4** flag settings are often appropriate when you are writing multi-user applications. Unless otherwise specified, the discussions in the following sections assume that these settings are being used. For details on the effects of changing these settings, please refer to last sections of this chapter.

- **Code4::accessMode = OPEN4DENY_NONE** (default) Files are opened in non-exclusive mode. This means that other applications can share the files with your application and have read and write access.
- **Code4::readOnly = 0** (default) Opens files in read/write mode. This allows you to read and write records to and from the data file.
- **Code4::readLock = 0** (default) There is no automatic record locking when a record is read.
- **Code4::lockAttempts = WAIT4EVER** (default) If a lock fails, CodeBase will keep retrying the lock until it succeeds.
- **Code4::lockEnforce = 1** An error is generated if an attempt is made to modify an unlocked record with a field function or **Data4::blank**, **Data4::changed**, **Data4::delete** or **Data4::recall**. This member variable must be set explicitly set to true (non-zero), since the default setting is false (zero).

Automatic Record Locking

Since locking and unlocking are time consuming operations (in the same order of magnitude as a write to disk), CodeBase functions that write a record to disk lock that record without unlocking it afterwards. This prevents redundant unlocking calls and allows you the option of leaving the record locked.

The only functions that modify records and perform automatic locking are: **Data4::append, Data4::appendBlank, Data4::flush,** and **Data4::write**.

Normally, you do not want to leave the record locked after these operations. To unlock the record, a call can be made to **Data4::unlock**:

```
dataFile.append( ) ;
dataFile.unlock( ) ;
```

Automatic Data File Locking

In addition to functions that automatically lock a data file record, there are CodeBase functions that automatically lock the entire data file. These are **Data4::memoCompress, Data4::pack, Data4::reindex, Data4::zap** and **Index4::reindex**. It is strongly recommended that not only that the data file be locked, but that the file be opened exclusively before performing these operations. Please refer to the section on **Code4::accessMode** in the CodeBase *Reference Guide*.

These functions leave the data file locked after they finish executing. As a result, if the data file was opened non-exclusively, these functions should be immediately followed by a call to **Data4::unlock**.

Automatic Unlocking Of Records

As a rule, CodeBase functions that move from an old record to a new record automatically remove any locks on the old record according to **Code4::unlockAuto**. These functions are **Data4::bottom, Data4::go, Data4::goEof, Data4::position, Data4::seek, Data4::seekNext Data4::skip,** and **Data4::top**. Refer to **Code4::unlockAuto** in the *Reference Guide* for more information on how the automatic unlocking works.

The only exception to this rule is when the new record is already locked. In that case, no unlocking is performed.

Common Multi-User Tasks

If you follow the advice about calling **Data4::unlock** after calling functions that automatically lock records and data files and are aware of functions that perform automatic unlocking, you should have little difficulty when writing multi-user applications.

To help you write multi-user applications, examples of common tasks performed by multi-user applications are provided below.

PROGRAM MULTI.CPP

Application "MULTI.CPP", which is perhaps simplistic, illustrates some basic operations which are present in almost any multi-user application: adding a record, modifying a record, finding a record, and reporting. It handles its multi-user aspects in a manner which is appropriate for many applications.

"MULTI.CPP" assumes that a data file "NAMES.DBF" is present along with a production index file containing a tag named "NAME". Data file "NAMES.DBF" also contains a field named "NAME".

```
#include "d4all.hpp"
extern unsigned _stklen = 10000 ; // for all Borland compilers

Code4 cb ; Field4 fieldName ;
Data4 dataNames ;
Tag4 tagName ;

void addRecord( void ) ;
void findRecord( void ) ;
void modifyRecord( void ) ;
void listData( void ) ;

void main()
{
    cb.accessMode = OPEN4DENY_NONE ;
    cb.readOnly = 0 ;
    cb.readLock = 0 ;
    cb.lockAttempts = WAIT4EVER ;
    cb.lockEnforce = 1 ;

    dataNames.open( cb, "NAMES" ) ;
    cb.exitTest(  ) ;

    fieldName.init( dataNames, "NAME" ) ;
    tagName.init( dataNames, "NAME" ) ;

    dataNames.top( ) ;

    for(;;)
    {
        cb.errorCode = 0 ;

        cout << endl << endl << "Record #: " << dataNames.recNo( )
            << "   Name: " << fieldName.str( ) << endl ;

        cout << "Enter Command ('a','f','l','m' or 'x') " ;

        int command =  getchar( ) ;
        switch( command )
        {
        case 'a':
            addRecord( ) ;
            break ;

        case 'f':
            findRecord( ) ;
            break ;

        case 'l':
            listData( ) ;
            break ;

        case 'm':
            modifyRecord( ) ;
            break ;

        case 'x':
            cb.closeAll( ) ;
            cb.initUndo( ) ;
            cb.exit( ) ;
        }
    }
}

void addRecord()
{
    char buf[100] ;
    printf( "Enter New Record\n" ) ;
    scanf( "%s", buf ) ;

    dataNames.appendStart( ) ;
    fieldName.assign( buf ) ;
    dataNames.append( ) ;
    dataNames.unlock( ) ;
}

void findRecord()
{
    char buf[100] ;
    printf( "Enter Name to Find\n" ) ;
    scanf( "%s", buf ) ;

    dataNames.select( tagName ) ;
    dataNames.seek( buf ) ;
}

void modifyRecord()
{
    int oldLockAttempts = cb.lockAttempts ;
    cb.lockAttempts = 1 ; // Only make one lock attempt

    rc = dataNames.lock(dataNames.recNo( ) ) ;
    if(rc == r4locked)
        cout << "Record locked. Unable to Edit" << endl ;
    else
    {
        char buf[100] ;
        cout << "Enter Replacement Record" << e ndl ;
        scanf( "%s", buf ) ;

        fieldName.assign( buf ) ;
        dataNames.flush( ) ;
        dataNames.unlock( ) ;
    }
    cb.lockAttempts = oldLockAttempts ;
}
void listData()
{
    cb.optStart( ) ;
    dataNames.optimize( OPT4ALL ) ;
    dataNames.select( tagName ) ;

    for( dataNames.top( ); ! dataNames.eof( ); dataNames.skip( ) )
```

```
        cout << dataNames.recNo( ) << " " << fieldName.str( ) << endl;

    dataNames.optimize( OPT4OFF ) ;
    cb.optSuspend( ) ;
}
```

Opening Files

It is especially important to check for any file open errors in multi-user applications because there is a chance that the file might be opened exclusively by another application.

Code fragment from MULTI.CPP

```
dataNames.open( cb, "NAMES" ) ;
cb.exitTest(  ) ;
```

Control Loops

Most data file editing software has some kind of loop where the end user can enter various editing and possibly reporting commands. The MULTI.CPP application is no exception. It allows you to 'add', 'find', 'list', 'modify' or 'exit'.

The start of the loop sets the CodeBase error code to zero. An error can occur if the user tries to modify a record before any have been added.

```
    dataNames.top( ) ;

    for(;;)
    {
        cb.errorCode = 0 ;

        cout << endl << endl << "Record #: " << dataNames.recNo( )
            << "   Name: " << fieldName.str( ) << endl ;

        cout << "Enter Command ('a','f','l','m' or 'x') " ;

        int command =  getchar( ) ;
        switch( command )
        {
        ...
        }
    }
```

Adding Records

In a single user situation, new records may simply be appended with a call to **Data4::append**. The multi-user situation is exactly the same, except that after appending the record **Data4::unlock** should be called (as in the *addRecord* function above). This call is necessary because the newly appended record remains locked after **Data4::append** completes.

Finding Records

The *findRecord* function is significant, from a multi-user perspective, more for what it does not do rather than what it does. Specifically, there is no multi-user logic necessary in this function. When **Code4::readLock** is false (zero), the data file functions do not perform any automatic locking as the data file is being read. Accordingly, you can search using index files and read data file records without having any extra multi-user logic present.

Code fragment from MULTI.CPP

```
void findRecord()
{
    char buf[100] ;
    printf( "Enter Name to Find\n" ) ;
    scanf( "%s", buf ) ;

    dataNames.select( tagName ) ;
    dataNames.seek( buf ) ;
}
```

Modifying Records

The **Field4** class is used to assign values to the fields, and when appropriate, the changes are flushed to disk. Explicit flushing can be accomplished by calling **Data4::flush.**

After flushing a modified record in a multi-user situation, an application should call **Data4::unlock.** When **Data4::flush** is called, the changed record gets written to disk. In order to accomplish this, **Data4::flush** locks the record and leaves it locked. Consequently, the call to **Data4::unlock** is suggested once the modified record is no longer required, in order to give other users an opportunity to modify the record.

The call to **Data4::flush** is not strictly necessary. This is because **Data4::unlock** is smart enough to recognize when the record buffer has changed. In this case, **Data4::unlock** will lock the record, flush the record, and then unlock the record. However, if the **S4OFF_MULTI** conditional compilation switch is used, it is necessary to call **Data4::flush** in order to immediately flush the record. This is because **Data4::unlock** does nothing when compiled with **S4OFF_MULTI** defined.

Code fragment from MULTI.CPP

```
void modifyRecord()
{
    int oldLockAttempts = cb.lockAttempts ;
    cb.lockAttempts = 1 ; // Only make one lock attempt

    rc = dataNames.lock(dataNames.recNo( ) ) ;
    if(rc == r4locked)
        cout << "Record locked. Unable to Edit" << endl ;
    else
    {
        char buf[100] ;
        cout << "Enter Replacement Record" << endl ;
        scanf( "%s", buf )  ;

        fieldName.assign( buf ) ;
        dataNames.flush( ) ;
        dataNames.unlock( ) ;
    }
    cb.lockAttempts = oldLockAttempts ;
}
```

Locking the Record before modifying

In the MULTI.CPP example, the record must be explicitly locked before it is modified by a field function, since the **Code4::lockEnforce** member is set to true (non-zero). This serves to ensure that only one application can edit a record at a time. The example sets the **Code4::lockAttempts** to **(int) 1**, to cause **Data4::lock** to immediately return **r4locked** if the record is already locked by another user, in which case the *modifyRecord* function is aborted. Adhering to this locking procedure will prevent the following undesirable scenerio from occurring.

Consider the case where two users of the application decide to modify the same record at the same time. When this happens, one application may write out its changes to disk before the second. The second application then writes to the disk, overwriting the first application's changes. Neither application knows that this has happened. User two makes changes based on an out of date version of the record, and user one loses all changes.

Multi-User Optimizations

This next section deals with using the memory optimization functions in a multi-user application. The optimizations can be used for both appending large quantities of records and for efficiently generating lists of records.

Listing Records

Function *listData* from the program MULTI.CPP is an example of reporting. Notice that memory optimization is turned on at the start of the function and turned off at the end of the function. This speeds up the report. In addition, when a network is being used it minimizes requests to the server.

In a multi-user application, the use of memory optimization is recommended when reporting but is strongly discouraged when editing. Refer to the "Memory Optimizations" chapter for additional information.

Notice that there are no unlocking function calls. As explained under "Finding Records" (above), when a record is read using the **Data4** class no automatic locking takes place. Consequently there is no need for any unlocking.

Code fragment from MULTI.CPP

```
void listData()
{
    cb.optStart( ) ;
    dataNames.optimize( OPT4ALL ) ;

    dataNames.select( tagName ) ;

    for( dataNames.top(); ! dataNames.eof(); dataNames.skip())
        cout << dataNames.recNo( ) << " "
            << fieldName.str( ) << endl;

    dataNames.optimize( OPT4OFF ) ;
    cb.optSuspend( ) ;
}
```

Repeated Appending

The next example multi-user application demonstrates how to append records from one data file to another.

PROGRAM APPEND.CPP

The program APPEND.CPP demonstrates how to append records from one data file to another. It assumes that data files "TO_DBF.DBF" and "FROM_DBF.DBF" are both present.

```
#include "d4all.hpp"
extern unsigned _stklen = 10000 ; // for all Borland compilers

void main()
{
    Code4 cb ;
    cb.optimize = OPT4ALL ;
    cb.optimizeWrite = OPT4ALL ;

    Data4 dataFrom( cb, "FROM_DBF"), dataTo( cb, "TO_DBF" ) ;
    cb.exitTest( ) ;

    Field4 infoFrom( dataFrom, "NAME" ), infoTo( dataTo, "FNAME" ) ;
    cb.optStart( ) ;

    cb.lockAttempts = 1 ;
    int rc1 = dataFrom.lockAll( ) ;
    int rc2 = dataTo.lockAll( ) ;

    if( rc1 != 0 || rc2 != 0 )
    {
        cout << "Locking Failed" << endl ;
        cb.exit( ) ;
    }

    for( int rc = dataFrom.top( ); rc == 0; rc = dataFrom.skip( ) )
    {
        dataTo.appendStart( ) ;
        infoTo.assignField( infoFrom ) ;
        dataTo.append( ) ;
    }

    dataFrom.unlock( ) ;
    dataTo.unlock( ) ;

    cb.closeAll( ) ;
    cb.initUndo( ) ;
}
```

In this application, both memory read optimization and memory write optimization are used. In order to do this, the **Code4::optimize** and **Code4::optimizeWrite** defaults are changed before the files are opened. Please refer to "Memory Optimizations" chapter for details.

Code fragment from APPEND.CPP

```
cb.optimize = OPT4ALL ;
cb.optimizeWrite = OPT4ALL ;

Data4 dataFrom( cb, "FROM_DBF"), dataTo( cb, "TO_DBF" ) ;
```

Note that **Code4::optStart** is called *after* the files are opened. This is because a call to **Code4::optStart** may use up the available memory. In this case, the calls to **Data4::open** could be slowed down as optimization was repeatedly suspended and re-invoked inside **Data4::open**. Internally, **Data4::open** calls **u4allocFree** when it allocates memory. When memory is not available, **u4allocFree** temporarily suspends memory optimization in an attempt to allocate the requested memory.

This example illustrates the only situation in which write optimization should be considered in a multi-user application. When writing to a file with write optimizations enabled, the file should be locked.

With the file locked and write optimization enabled, the repeated appending goes considerably faster. The trade off is that if any other application attempts to read the records being appended, the other application could, at worst, generate CodeBase errors or could read garbage information. To avoid this, you can open the files exclusively or not use write optimization.

In this specific case, the read optimization is a very good idea. This is because there is no chance that information being returned could be out of date because the file is locked. As a result of the read optimization, the speed improvements in this example are considerable.

This application skips sequentially through data file "FROM_DBF" using memory optimization. Consequently, it is important to use function **Data4::skip** rather than **Data4::go** because function **Data4::skip** detects the sequential reading and does special performance optimizations.

Since the data files are locked going into the loop, they stay locked throughout the entire loop with no automatic unlocking occurring.

Code fragment
from APPEND.CPP

```
for( int rc = dataFrom.top( ); rc == 0 ; rc =dataFrom.skip( ) ) {
    dataTo.appendSta rt( ) ;
    infoTo.assignField( infoFrom ) ;
    dataTo.append( ) ;
}
```

Once the records have been appended, the files are unlocked and closed. When data file "TO_DBF.DBF" is unlocked its changes are automatically flushed to disk.

Code fragment from APPEND.CPP

```
dataFrom.unlock( ) ;
dataTo.unlock( ) ;

cb.closeAll( ) ;
```

The calls to **Data4::unlock** are not strictly necessary in this example because **Code4::closeAll** does flushing and unlocking automatically. Alternatively, **Code4::unlock** could have been called to unlock both files at once.

Avoiding Deadlock

When locking multiple data files at once, you need to be careful to avoid deadlock. Deadlock happens when one application waits for a second application to unlock something and the second application waits for the first application to unlock something else. Since they are both waiting for each other, they both wait forever. This program avoids deadlock by changing **Code4::lockAttempts** from the default of unlimited retries to one try. If the lock attempt fails, the program exits.

Code fragment from APPEND.CPP

```
cb.lockAttempts = 1 ;
int rc1 = dataFrom.lockFile( ) ;
int rc2 = dataTo.lockFile( ) ;

if( rc1 != 0 || rc2 != 0 )
{
    cout << "Locking Failed" << endl ;
    cb.exit( ) ;
}
```

Another way to avoid deadlock is to be careful to always lock data files in the same order. If one application locks data file "FROM_DBF" before "TO_DBF" using unlimited retries, then all applications using unlimited retries should do the same.

If applications always lock data files in the same order, deadlock is not possible. This is because once the first application succeeds in locking the first data file, other applications will wait for the first application to finish its locking, do what it needs to do and then unlock all of the data files. The best way to ensure you are always locking data files in the same order is to lock and unlock all data files together.

Group Locks

Sometimes it is desirable to lock many items as a group. In this case, either all items are locked or no items are locked. This functionality will help minimize the chance of deadlock. CodeBase supplies functions that will put an item on a queue so that the whole queue can be locked as a group. The functions **Data4::lockAdd**, **Data4::lockAddAppend**, **Data4::lockAddFile**, **Data4::lockAddAll** and **Relate4set::lockAdd** put their respective items on a queue to be locked. These functions do NOT lock any items. The items are locked by a call to **Code4::lock**. If any item in the queue fails to be locked, the successful locks are removed, **Code4::lock** returns **r4locked** and the queue remains intact for a future lock attempt by a subsequent call to **Code4::lock**. When **Code4::lock** succeeds, all the locks are in place and the queue is emptied. Refer to **Code4::lock** in the *Reference Guide* for more details.

Exclusive Access

The simplest types of multi-user applications are those that open all of their files in exclusive access mode. If any other application tries to open a file which has been opened exclusively by another user, the application gets a file open error.

The big disadvantage of this method is that no other applications can use files that you are using. This is not a recommended method of creating multi-user applications.

The main use of opening files exclusively is when you are performing packing, zapping, indexing or reindexing. This prevents other applications from generating errors or garbage output if they use the file while one of these activities is occurring.

Opening Files Exclusively

Whether a file is opened exclusively is determined by the current status of the **Code4::accessMode** flag. This flag specifies what access OTHER users have to the current file. **Code4::accessMode** has three possible values:

- **OPEN4DENY_NONE** Open the database files in shared mode. Other users have read and write access. This is the default value.
- **OPEN4DENY_WRITE** Open the database files in shared mode. The application may read and write to the files, but other users may only open the file in read only mode and may not change the files.
- **OPEN4DENY_RW** Open the database files exclusively. Other users may not open the files.

This flag can be changed any time after the **Code4** object has been initialized. Changes to the flag only effect those files that are opened thereafter. Therefore if the access mode must be altered for an open file, the file must be closed and then reopened.

```
cb.accessMode = OPEN4DENY_RW;
dataFile.open( cb, "DATAFILE" ) ; // open exclusively
```

If a file cannot be opened exclusively due to the fact that some other application already has it open, the file open fails. When this happens, and the **Code4::errOpen** flag is set to true , an error message is generated.

Read Only Mode

If your application reads information from a file and never modifies it, you can open the file in read only mode. This mode must be used when you only have read access to a file. This is a common situation on a network where a database is shared among many users, but only a few users have the authority to make changes to it.

Opening Files In Read Only Mode

Whether a file is opened in read only mode is determined by the current status of the **Code4::readOnly** flag. If the flag is set to its default value of false, files are opened in read / write mode. If this flag is set to true (non-zero), files are opened in read only mode.

Lock Attempts

When a lock is performed on a file, the possibility exists that some other application has already locked that file or a portion of the file. When this occurs, what CodeBase does next is determined by the status of the **Code4::lockAttempts** flag.

The default value of **Code4::lockAttempts** is **WAIT4EVER**. This indicates that CodeBase will keep trying to establish a lock until it succeeds. The advantage of this method is that it simplifies programming because the application can assume that all lock attempts succeed. Unfortunately, this can increase the chances of deadlock bugs being present in an application. In addition, if applications attempt locking for long periods of time, users can be left waiting. One potential solution is to use the **S4LOCK_HOOK** conditional compilation switch and put the number of retries directly under the control of the end user. Refer to the CodeBase *Reference Guide* "Conditional Compilation Switches" chapter for more information on this technique.

Code fragment from MULTI.CPP

```
cb.lockAttempts = WAIT4EVER; // Retry forever
```

If **Code4::lockAttempts** is set to any value greater than one, CodeBase keeps trying the lock at approximately one second intervals until either a lock is established or the number of attempts equals **Code4::lockAttempts**. If the lock fails, a value of **r4locked** is returned from the function attempting the lock.

If you wish to perform a special operation if the lock fails, such as displaying a message to the user, you should set **Code4::lockAttempts** to one and test for return values.

Valid settings for **Code4::lockAttempts** are **WAIT4EVER** (-1) or any value greater than or equal to 1. Any other value is undefined.

Automatic Record Locking

When **Code4::readLock** is true, several CodeBase functions automatically lock records before reading them. This ensures that no other application can modify a record that your application has in its record buffer.

Unfortunately, locking data file records when they are only being read can create unnecessary locking contention. Simply put, this would increase the chance of a record being locked when an end user just wants to look at it. Consequently, the default of no automatic read locking is appropriate for most multi-user applications.

The following list of functions perform record locking when the **Code4::readLock** flag is set to true: **Data4::bottom, Data4::go, Data4::position, Data4::seek, Data4::seekNext, Data4::skip,** and **Data4::top**.

Enforced Locking

Locking a record before it is modified is very important in the multi-user configuration, since it ensures that only one application can edit a record at a time. The **Code4** member variable **Code4::lockEnforce** can be used to ensure that an application has explicitly locked a record before it is modified with a field funcition or the following data file functions: **Data4::blank**, **Data4::changed**, **Data4::delete** or **Data4::recall**. When **Code4::lockEnforce** is set to true (non-zero), an **e4lock** error is generated when a attempt is made to modify an unlocked record.

An alternative method of ensuring that only one appliction can modify a record at a time is to deny all other applications write access to a data file. Write access can be denied to other applications by setting **Code4::accessMode** to **OPEN4DENY_WRITE** or **OPEN4DENY_RW** before opening the file. Thus, it is unnecessary to explicitly lock the record before editing it, even when **Code4::lockEnforce** is true (non-zero), since no other application can write to the data file. This method suffers from the same disadvantage as discussed in "Exclusive Access" section of this chapter. The restricted access to the files prevents the creation of practical multi-user applications.

12 Performance Tips

Programmers are always looking for ways to improve the performance of their applications. This chapter provides the programmer with tips on how to achieve the highest performance when building applications with CodeBase.

Memory Optimization

The prudent use memory optimization can enhance the performance of an application. For details on when to use memory optimization refer the "Memory Optimizations" chapter of this guide. Write optimization is most useful when making multiple updates on files, such as appending many records at once. In general, when memory optimization is used, **Data4::skip** is faster than **Data4::go**.

Memory Requirements

The **Code4** member variable **Code4::memStartMax** specifies the maximum amount of memory allocated for memory optimization. Theoretically, the more memory that CodeBase can use for memory optimization the greater the improvement in performance. In practise, if **Code4::memStartMax** is too large, CodeBase may not be able to allocate all of the requested memory causing the memory optimization to work less efficiently. In some operating systems all requested memory is allocated; but as virtual memory, which is counter-productive to memory optimization. On the other hand, if **Code4::memStartMax** too small, the potential benefits of memory optimization will not warrant the overhead.

Observing the performance improvement

The improvement in performance can be observed by using **Code4::optAll**. **Code4::optAll** ensures that memory optimization is fully implemented by locking and fully read and write optimizing all opened data, index and memo files. A comparison of the performance can then be made between the application that employs memory optimization with one that does not.

Functions that can reduce performance

Memory optimization acts by buffering portions of files in memory and thus decreases the number of disk accesses and improves performance. Therefore, frequent calls to CodeBase functions that read data from disk will negate the effects of read optimization. For this reason, functions such as **Data4::refresh**, **Data4::refreshRecord** and **File4::refresh** should not be called repeatedly while memory optimization is invoked. Frequent flushing also reduces the benefits of write optimization.

Locking

Locking can take as long a disk access, so repeated record locks are inefficient. It is more efficient to lock an entire file when multiple updates are necessary. The best way to lock a file that requires many modifications is to use **Data4::lockAddAll** with **Code4::lock,** or call **Data4::lockAll.**

A **Code4** setting that can influence performance is **Code4::readLock**. When **Code4::readLock** is set to true (non-zero), each record is locked before it is read, which can reduce performance. It is useful to set **Code4::readLock** to true when the records are to be modified.

Another important **Code4** variable that can affect performance in a network environment is **Code4::lockDelay**, which determines how long to wait between lock attempts. Consider this variable carefully because if the value is too small, it can cause an increase in network traffic and possibly a decrease network performance.

Appending

Whenever possible, it is more efficient to append many records at once rather than one at a time. When appending many records, it is sometimes faster to close the index files first, append the records and then reindex. The files must be locked before the records may be appended and the most efficient method is to call **Data4::lockAddAll** with **Code4::lock,** or **Data4::lockAll,** as discussed above. Batch appending can also take advantage of write optimization, which will improve performance.

Time Consuming Functions

Certain CodeBase functions are inherently time consuming and this aspect should be considered when incorporating them into an application. Avoid calling time consuming functions repeatedly, since this can reduce performance. Some functions are meant to be used for debugging purposes and should not be used in the final user application.

The following functions reindex index files, which is a time consuming procedure, so they should be used with care. **Data4::reindex** and **Index4::reindex** can be called to explicitly reindex files. Both **Data4::pack** and **Data4::zap** automatically reindex the associated index files when they are called.

Data4::pack and **Data4::zap** are time consuming functions even when there are no tags to reindex. These functions delete records from a data file and then reorganize the remaining records, which is a time consuming process.

Data4::check is a function that determines whether an index file has been corrupted, which is useful for debugging. This function can take as long as reindexing and therefore application performance can be hindered.

Queries and Relations

CodeBase uses Query Optimization to greatly increase the performance of queries in relation sets. To ensure that the Query Optimization can be used by CodeBase, the query expression must have a corresponding tag expression. See the "Query Optimization" chapter in this guide for more details on how to ensure that the Query Optimization will be invoked.

Another important issue to consider when manipulating relations is how to specify the sort order of the query set. The most efficient manner will depend on the size of the database. There are three ways in which a sorted order may be specified for the query set.

1. The sorted order can be determined by the selected tag in the master data file.
2. If there is no selected tag for the master data file, then the natural order of query set is used.
3. The function **Relate4set::sortSet** can be used to specify the sorted order.

If there is a tag that specifies the same sort order as a **Relate4set::sortSet** expression, then use the tag sorted order when the query set is almost the same size as the database. If the query set is almost the same size as the database, then **Relate4set::sortSet** is inefficient when compared with the tag sorted order or the natural order. Use **Relate4set::sortSet** when the query set is small compared to the size of the database. This will result in better performance when compared with the selected tag ordering.

If there is no tag that specifies a desired sort order, then use **Relate4set::sortSet** to sort the query set. Using **Relate4set::sortSet** will be faster than creating a new tag to specify the sort order, regardless of the size of the query set.

General Tips

The following section discusses miscellaneous issues that may influence the performance of an application.

- Be sure to use the **Field4/Str4** functions to manipulate fields for the best results. Avoid using the **Expr4** module to perform calculations on fields, otherwise the application will execute at a slower rate.
- It is recommended that all necessary tags be constructed when the index file is created with either **Data4::create** or **Index4::create**. This method is more efficient than adding new tags by calling **Index4::tagAdd** or **Tag4info::add** later in the application.
- The **Mem4** module specializes in repeated allocation and freeing of fixed length memory. As a result, it is better than some operating systems at memory management. Therefore, it is advantageous to use this module whenever possible.
- File access is quicker when the file is opened exclusively, since

no future record or file locks will be necessary.

- The **Code4** member variables that determine how many memory blocks are allocated should be defined at the beginning of the program and should not be changed during the program. This allows CodeBase to share memory pools efficiently. Set the following **Code4** settings at the beginning of the application.

  Code4::memStartData
  Code4::memStartIndex
  Code4::memStartBlock
  Code4::memStartLock
  Code4::memStartMax
  Code4::memStartTag

13 Transaction Processing

At times it is necessary to have a group of actions executed as a unit. In this case, either all actions are completed or none of the actions are completed.

To illustrate this, consider the everyday example of transferring funds from one bank account to another. There are two steps to this process: first one account has the money debited and then the second account has the sum credited. A failure could occur after the debit but before the credit, in which case the money would be removed from the first account but the second account remains unchanged, resulting in lost funds. In this type of situation, one would like an all or nothing response, where either both the debit and credit are completed or nothing happens and the accounts remain unchanged.

Transactions

A transaction is one way of treating a group of actions as a whole. The scope of a transaction is delimited by a specified "start" and "end" between which the actions are treated as a unit. A transaction has the all or nothing property, so that either all the actions within the scope of a transaction are completed or none are completed. When a transaction has successfully completed all the operations within its scope, it is "committed". A transaction can be aborted and all the changes that were made up until the failure can be reversed. This constitutes a "rollback". After the transaction has terminated, the changes are permanent and can not be reversed.

PROGRAM TRANSFER.CPP

The following program shows how a simple bank transfer can be accomplished using a CodeBase transaction.

```
//TRANSFER.CPP

#include "d4all.hpp"

extern unsigned _stklen = 10000 ; //for all Borland Compilers

Code4 codeBase ;
Data4 datafile ;

Field4 acctNo, balance ;
Tag4 acctTag, balTag ;

void OpenLogFile( void )                   //Opening or creating a log file is
                                           //only required for STAND-ALONE mode.
{
    int rc = codeBase.logOpen( 0, "user1") ;       //NULL is passed as the log file name
    if ( rc == r4noOpen )                          //so the default "C4.LOG" is used as
    {                                              //the log file name.
        codeBase.errorCode = 0 ;
        codeBase.logCreate( 0, "user1" ) ;
    }
}

void OpenDataFile( void )
{
    datafile.open( codeBase, "BANKDATA.DBF" ) ;

    acctNo.init( datafile, "ACCT_NO" ) ;
    balance.init( datafile, "BALANCE" ) ;

    acctTag.init( datafile, "ACCT_TAG" ) ;
    balTag.init( datafile, "BAL_TAG" ) ;
}

int Credit( long toAcct, double amt )
{
    datafile.select( acctTag ) ;
    int rc = datafile.seek( toAcct ) ;

    if (rc != r4success ) return rc;

    double newBal = double( balance ) + amt ;
    balance.assignDouble( newBal ) ;
    return r4success;
}

int Debit( long fromAcct, double amt )
{
    return Credit(  fromAcct, -amt ) ;
}

void Transfer( long fromAcct, long toAcct, double amt )
{
    codeBase.tranStart( ) ;

    int rc1 = Debit( fromAcct, amt ) ;
    int rc2 = Credit( toAcct, amt );

    if (rc1 == r4success && rc2 == r4success)
        codeBase.tranCommit( ) ;
    else
        codeBase.tranRollback( ) ;
}

void PrintRecords( void )
{
    for( int rc = datafile.top(); rc == r4success; rc = datafile.skip())
    {
        cout <<"----------------------------------" << endl
             <<"Account Number: "<< (long) acctNo    << endl
             <<"Balance       : "<< (double) balance << endl ;
    }
    cout <<"========================================"<< endl ;
}

void main( void )
{
    codeBase.errOpen = 0 ;
    codeBase.safety = 0 ;
    codeBase.lockAttempts = 5 ;

    OpenLogFile() ;

    OpenDataFile() ;

    PrintRecords() ;

    // The account number 56789 doesn't exist in the database,
    // the transfer is aborted and database is not affected

    Transfer( 12345, 56789, 200.00 ) ;
    PrintRecords() ;

    // Both accounts exist so the transfer is completed
    // and the database is updated

    Transfer( 12345, 55555, 150.50 ) ;
    PrintRecords() ;

    codeBase.initUndo( ) ;
}
```

In the TRANSFER.CPP program, a transfer takes place when there is a debit from one account followed by a credit to another, as shown by the following function calls.

Code fragment for TRANSFER.CPP

```
int rc1 = Debit( fromAcct, amt ) ;
int rc2 = Credit( toAcct, amt );
```

These two operations must both be successful in order for the transfer to succeed. Therefore, these two actions must be contained within a transaction where either both the debit and the credit succeed or neither will be completed and the database will not be changed. In CodeBase, a transaction is delimited by the **Code4** member functions **Code4::tranStart** and **Code4::tranCommit**, which initiate and terminate the transaction respectively.

```
codeBase.tranStart() ;

...   //code to be treated as a unit

codeBase.tranCommit() ;
```

Events may occur that cause failure during a transaction. For example, if there is a power failure or the hardware crashes during a transaction, one would want to be able to reverse any changes that had occurred before the failure. Programmers must anticipate and test for possible failure points within a transaction. For instance, the example program TRANSFER.CPP checks to see if both specified accounts exist in the database and a failure occurs if either account did not exist. When the program encounters a failure of this sort, it calls **Code4::tranRollback** to abort the transaction and restore the database to its original state.

If the transfer succeeds, the transaction is completed by **Code4::tranCommit**, which commits the changes to the database. After the transaction has been committed, **Code4::tranRollback** can NOT be used restore the database to the state that existed before the transaction was started. An error is generated if **Code4::tranRollback** is called after **Code4::tranCommit**.

Code fragment from TRANSFER.CPP

```
if (rc1 == r4success && rc2 == r4success)
        codeBase.tranCommit() ;
    else
        codeBase.tranRollback() ;
```

How a Transaction works

CodeBase has implemented transactions in the following manner. When a transaction is initiated with **Code4::tranStart**, all the changes to the database are recorded in a log file. The log file stores both the old value and the new value, as well as who made the change. At every step during a transaction, each change is recorded in the log file and then the database is modified. After the transaction has been committed using **Code4::tranCommit,** the changes to the database are permanent. During a rollback, **Code4::tranRollback** uses the original values from the log file to reverse all the changes made to the database.

Logging in the Stand-Alone Case

All modifications that are made while an application is running, which are outside the scope of a transaction, can be automatically recorded in the log file. Automatic logging is useful when a back up copy of all changes is required. There are times when logging is not necessary; for example, when copying data files. In the stand-alone configuration, the logging can be turned off to save time and disk space. The following discussion applies only to logging in the stand-alone configuration.

Code4::log The **Code4** member variable **Code4::log** is used to specify whether changes to the data files are automatically recorded in a log file. Changing this setting only affects the logging status of the files that are opened subsequently. The files that were opened before the setting was changed retain the logging status that was set when the file was opened. **Code4::log** has three possible values:

- **LOG4ALWAYS** The automatic logging takes place for all the data files for as long as the application is running. The logging can NOT be turned off for the current data file by calling **Data4::log**.
- **LOG4ON** The automatic logging takes place for all the data files for as long as the application is running. In this case, **Data4::log** can be used to turn the logging off and on for the current data file. CodeBase uses this value as the default.
- **LOG4TRANS** Only the changes made during a transaction are automatically recorded in the log file. Logging may be turned on and off for the current data file by calling **Data4::log**.

Data4::log When a file is opened with **Code4::log** set to either **LOG4ON** or **LOG4TRANS, Data4::log** can be used to turn the logging off and on for the current data file. **Data4::log** only has an effect on logging while the data file is open and it has no effect the logging that is done during a transaction.

Pass false (zero) to **Data4::log,** to turn the logging off and pass it true (non-zero) to turn the logging back on. **Data4::log** also returns the previous logging status. If void is passed to **Data4::log**, the current logging status is returned. The possible return values are as follows :

- **r4logOn** This return value means that logging is turned on for the data file.
- **r4logOff** This return value means that either the logging is turned off for the data file or that logging in general has not been enabled.
- **< 0** An error has occurred.

turning logging off

```
codeBase.log = LOG4ON ; //Logging of changes takes place
                          //for all data files opened subsequently

datafile.open( codeBase, "DATA.DBF" ) ;

rc = datafile.log( 0 ) ; //logging is turned off for this data
                            //file, the previous logging status is
                            //returned
rc = datafile.log( ) ;  //returns the current logging status
```

Log files

In the stand-alone configuration, a log file must exist or be explicitly created before any automatic logging or transaction logging can take place. If a log file does not exist, **Code4::tranStart** will generate an error.

The log file may be manually opened or created by using **Code4::logOpen** and **Code4::logCreate** respectively. **Code4::logOpen** and **Code4::logCreate** both take the name of the log file and user identification as parameters. If NULL is passed as the file name, then the default file name "C4.LOG" is used. **Code4::logOpen** and **Code4::logCreate** must be called before **Data4::open** or **Data4::create**.

Generally, one would like to open a log file if it exists, otherwise create a new log file. This concept is the same as discussed in the "Opening or Creating" section of the "Database Access" chapter of this guide.

Code fragment from TRANSFER.CPP

```
void OpenLogFile( void )
{
    int rc = codeBase.logOpen( 0, "user1") ;
    if ( rc == r4noOpen )
    {
        codeBase.errorCode = 0 ;
        codeBase.logCreate( 0, "user1" ) ;
    }
}
```

If **Code4::logOpen** or **Code4::logCreate** have not been explicitly called, **Data4::open, Data4::create** and **Code4::tranStart** automatically try to open a log file by calling **Code4::logOpen**. **Code4::logOpenOff** can be used to instruct **Data4::open**, **Data4::create** and **Code4::tranStart** not to automatically open the log file. When **Code4::logOpenOff** is used, no transactions can start unless the log file is opened explicitly.

the log file is not automatically opened

```
rc = codeBase.logOpenOff( ) ;
datafile.open( codeBase, "DATA.DBF" ) ;
```

Logging in the Client-Server case

In the client-server configuration, the automatic logging and transaction logging is handled by the server. Calling the functions **Code4::logCreate, Code4::logOpen** and **Code4::logOpenOff** by a client application will have no effect on the log file, since it is controlled by the server. These functions will always return **r4success** when called by a client application.

Locking

Every time a data file is being modified one must consider which locking procedure is appropriate. Locking is necessary step in perserving the integrity of the database. CodeBase has both automatic and explicit locking features, as discussed in the "Multi-user Applications" chapter in this guide.

Automatic locking during a transaction

While a transaction is in progress, any automatic locking that is required during a transaction is performed, but CodeBase acts as though the **Code4::unlockAuto** is set to **LOCK4OFF,** so no automatic unlocking will occur.

Automatic locking and unlocking can occur when a transaction is commited or rolled back. In both cases, record buffer flushing may be required. The locking and unlocking procedure follows that of **Data4::flush**. If a new lock is needed, everything is unlocked according to **Code4::unlockAuto** and then the lock is placed. Otherwise, nothing is unlocked when no new locks are required. It may be necessary to explicitly unlock files after the transaction is commited or rolled back depending on the **Code4::unlockAuto** setting.

Automatic locking vs. Explicit locking

Whether using the automatic or explicit locking one must decide on a value for the **Code4** member variable **Code4::lockAttempts**. This member has a default value of **WAIT4EVER**, which may result in deadlock in a multi-user environment. To prevent deadlock, set **Code4::lockAttempts** to a reasonable finite number.

The example program TRANSFER.CPP sets **Code4::lockAttempts** to a finite value and takes advantage of the automatic locking properties of CodeBase. When **Data4::seek** is called during the transaction, the current record may need to be flushed before the new record can be loaded into the recorded buffer. Normally, if a new lock is required, everything is unlocked according to **Code4::unlockAuto** and then the lock is placed, but the seek takes place during a transaction so no unlocking is performed.

Code fragment from TRANSFER.CPP

```
codeBase.lockAttempts = 5 ;
```

When an application is running in a multi-user environment, it is strongly recommended that records be locked before they are modified to prevent more than one user access to the record at the same time. In this case, it is prudent to set **Code4::lockEnforce** to true (non-zero) to ensure that a record is explicitly locked before it is modified. If it is known before hand that many changes will be made, use **Data4::lockAll,** or use **Data4::lockAddAll** followed by **Code4::lock** to explicitly lock the files.

If the file is explicitly locked, it must be explicitly unlocked by calling **Code4::unlock** or **Data4::unlock** after the transaction is committed or rolled back. If an attempt is made to unlock files during a transaction, an error is generated.

Whether locking is done explicitly or implicitly, a lock may not succeed because another application has that particular file or record locked, in which case **r4locked** is returned. To circumvent this, build the CodeBase library with the conditional compilation switch **S4LOCK_HOOK** defined. This enables CodeBase to automatically execute the function **code4lockHook** when a lock has failed. **code4lockHook** is a function that is defined by the application programmer, so that it can allow the end user to retry the lock.

Refer to the "Conditional Compilation Switches" chapter of the CodeBase *Reference Guide* for more information on **S4LOCK_HOOK** and **code4lockHook**.

14 Copying CodeBase Classes

C++ provides a default class assignment operator (=) for copying objects of the same type. This operator automatically makes a copy of every member variable in the class object. This convenience, however, leads to some problems when used with the CodeBase classes. When dealing with objects containing linked lists and dynamically allocated memory, for instance, a copy of an object can lead to memory corruption and corruption of the objects.

A copy of an object is also made when it is passed by value to a function. This often overlooked feature can lead to some of the most obscure application bugs. See the "C++ Programming" chapter in this guide for more information about passing variables as pointers, by value, and by reference.

The following chart lists all the CodeBase classes and the action(s) taken when an object is copied with the assignment (=) operator or by passing a copy of an object to a function. The classes in italics may not be copied.

Class	Discussion
Code4	*Since the **Code4** class contains several linked lists and pointers to allocated memory, copying a **Code4** object is not allowed. A linker error results when an attempt is made to copy this object.*
Data4	Since the **Data4** class simply contains a pointer to internally allocated memory, **Data4** objects may be copied freely. Since a copied object points to the same information as the original, any changes (positioning or otherwise) made with the copy effect the original.
Date4	The date stored in the original object is copied into the copy. Modifications made to the copy do not alter the original.
Expr4	**Expr4** objects may be copied freely since they merely contain a pointer to internally allocated memory. Changes made to a copy are reflected in the original.
Field4	**Field4** objects may be copied freely since they merely contain a pointer into the database's record buffer. Changes made to a copy are reflected in the record buffer and in the original object.

Field4memo	**Field4memo** objects may be copied freely since they contain a pointer to internally allocated memory containing the memo (or record buffer if the field is not a memo). Changes in the original are reflected in the copy.
Field4info	*Copying **Field4info** objects is not allowed, since they contain pointers to dynamically allocated memory. Attempts to copy these objects result in linker errors.*
File4	**File4** objects may be copied, but since both objects contain the same file handle, caution should be exercised that neither object be accessed once the file has been closed. When memory optimizations are used, **File4** objects should not be copied.
File4seqRead	***File4seqRead** objects may not be copied. Since **File4seqRead** objects contain information specific to the buffering of the file, copying the object can result in corrupted information being retrieved from disk. Attempts to copy this class results in linker errors.*
File4seqWrite	***File4seqWrite** objects may not be copied. Since **File4seqWrite** objects contain information specific to the buffering of the file, copying the object can result in corrupted information being written to disk. Attempts to copy this class results in linker errors.*
Index4	**Index4** objects may be freely copied. Since they contain a pointer to internally allocated memory, changes made in a copy of an object are reflected in the original object.
List4	*Copying a **List4** object is not allowed, since adding and deleting nodes from a copy corrupt the original. Attempts to copy **List4** objects result in linker errors.*
Mem4	**Mem4** objects may be freely copied. Since they contain a pointer to internally allocated memory, changes made in a copy of an object are reflected in the original object.
Relate4	**Relate4** objects may be copied freely since they merely contain a pointer to internally allocated memory. Changes made to a copy alter the original.
Relate4iterator	**Relate4iterator** objects may be copied freely since they merely contain a pointer to internally allocated memory. Changes made to a copy alter the original.
Relate4set	**Relate4set** objects may be copied freely since they merely contain a pointer to internally allocated memory. Changes made to a copy effect the original.
Sort4	***Sort4** objects may not be copied, since they contain internal linked lists, dynamically allocated memory and file handles which are corrupted when accessed using a copy. Attempts to copy **Sort4** objects result in linker errors.*

Str4	Since **Str4** is a pure virtual class, no objects may be constructed or copied.
Str4char	***Str4char*** *objects may be freely copied. Since* ***Str4char*** *contains its own internal static memory, changes made to a copy do not alter the original.*
Str4flex	**Str4flex** objects may be copied with the **Str4flex::operator =**, or the **Str4flex::Str4flex(Str4flex &)** constructor. This operator overloads the default C++ class assignment operator to ensure that dynamically allocated memory is properly allocated and freed by the objects.
Str4large	**Str4large** objects may be freely copied. Since **Str4large** contains its own internal static memory, changes made in a copy of an object do not alter the original object.
Str4len	**Str4len** objects may be freely copied. Since **Str4len** objects do not contain their own allocated memory, changes made to the copy are reflected in the original and in the information referenced by the **Str4len** object.
Str4max	**Str4max** objects may be freely copied. Since **Str4max** objects do not contain their own allocated memory, changes made to the copy alter the original and the information referenced by the **Str4max** object.
Str4ptr	**Str4ptr** objects may be freely copied. Since **Str4ptr** objects do not contain their own allocated memory, changes made to the copy alter the original and the information referenced by the **Str4ptr** object.
Str4ten	**Str4ten** objects may be freely copied. Since **Str4ten** contains its own internal static memory, changes made to a copy do not alter the original.
Tag4	**Tag4** objects may be copied freely. Changes made to a copy are reflected in the original.
Tag4info	***Tag4info*** *objects may not be copied, since they contain pointers to dynamically allocated memory which are corrupted when accessed with a copy. Attempts to do so result in linker errors.*

It is mentioned above that passing an object to a function by value creates a copy of the object. Several classes do not allow copies to be made and so they may not be passed by value. If objects of these classes must be accessible in other functions, they may be passed by reference. Below is an incorrect example which tries to pass a **Code4** and a **Data4** object to a function:

```
#include "d4all.hpp"

int openFileEx( Code4 codeBase, Data4 data, char *name )
{
    int oldAccessMode = codeBase.accessMode ;
    codeBase.accessMode = OPEN4DENY_RW ;

    data.open( codeBase, name ) ;

    codeBase.accessMode = oldAccessMode ;

    return data.isValid( ) ;
}

void main( )
{
    Code4 cb ;
    Data4 db ;
    openFileEx( cb, db, "INFO" ) ;
    cb.closeAll( ) ;
    cb.initUndo( ) ;
}
```

This code sample generates a linker error indicating **Code4::Code4(Code4 &)** is unresolved. This error is generated as a result of passing a copy of **Code4** instead of a reference. The copy of **Data4** , being valid, generates no error messages. The program compiles successfully if the declaration of *openFileEx* is defined as the following:

```
int openFileEx( Code4 &codeBase, Data4 data, char *name)
```

In this example, the **Data4** object is passed by value instead of by reference. When the object is passed, only four bytes, which is the size of a pointer, are actually copied. Compilers also generate slightly more efficient code for accessing a variable passed by value, than for those passed by reference.

15 Mixing CodeBase C++ API & CodeBase C API

Applications written with the CodeBase C API may be integrated with the CodeBase C++ API to take advantage of the C++ capabilities. This compatibility makes improvements on large CodeBase C API applications a painless venture.

Since the CodeBase C++ API is based upon the structures of CodeBase C API, code written using either product may be mixed, following these few steps.

- Change the `#include "d4all.h"` to `"d4all.hpp"`. This not only includes the C++ class definitions, but also includes **d4all.h** with all the C prototypes and structure definitions.
- Convert the application's **CODE4** variable to a **Code4** object.
- Remove the application's call to **code4init**. The **Code4** constructor automatically performs initialization.
- If the application's CodeBase C API structure variables are needed in the C++ additions, or if the CodeBase C API functions are called with CodeBase C API classes, refer to the following chart.

C Structure Name	Class Name	C Structure Usage From C++ Object	Object Usage From C Structure
CODE4	Code4	Code4 object	Rename to class
DATA4	Data4	Data4.data	Data4::Data4(DATA4 *)
EXPR4	Expr4	Expr4.expr	Expr4::Expr4(EXPR4 *)
FIELD4	Field4	Field4.field	Field4::Field4(FIELD4 *)
FIELD4INFO	Field4info	Field4info.fields( )	n/a
FIELD4	Field4memo	Field4memo.field	::Field4memo(FIELD4 *)

FILE4	File4	File4 object	Rename to class
FILE4SEQ_READ	File4seqRead	File4seqRead object	Rename to class
FILE4SEQ_WRITE	File4seqWrite	File4seqWrite object	Rename to class
INDEX4	Index4	Index4.index	Index4::Index4(INDEX4 *)
LIST4	List4	List4 object	Rename to class
MEM4	Mem4	Mem4.mem	Mem4::Mem4(MEM4 *)
RELATE4	Relate4	Relate4.relate	::Relate4(RELATE4 *)
RELATE4	Relate4iterator	Relate4.relate	Relate4iterator::operator=
RELATE4	Relate4set	Relate4.relate	::Relate4set(RELATE4 *)
SORT4	Sort4	Sort4 object	Rename to class
TAG4	Tag4	Tag4.tag	Tag4::Tag4(TAG4 *)
TAG4INFO	Tag4info	Tag4info.tags( )	n/a

Example The following simple program uses CodeBase C API to create a database.

```
#include "d4all.h"
extern unsigned _stklen = 10000 ;

FIELD4INFO fields[ ] =
{
    { "NAME", 'C', 20, 0 },
    { "AGE", 'N', 3, 0 },
    { 0,0,0,0 },
} ;
void main( )
{
    CODE4 cb ;
    DATA4 *data ;

    code4init( &cb ) ;
    cb.safety = 0 ;

    data = d4create( &cb, "PERSON", fields, 0 ) ;

    if( data )
        printf( "PERSON data file successfully created.\n" ) ;

    code4close( &cb ) ;
    code4initUndo( &cb ) ;
}
```

If this program were to be modified to also create an index file, but the changes were to be made with CodeBase C++ API, it might look something like the following (The bold lines are the changed/new lines.)

```
#include "d4all.hpp"
extern unsigned _stklen = 10000 ;

FIELD4INFO fields[ ] =
{
    { "NAME", 'C', 20, 0 },
    { "AGE", 'N', 3, 0 },
    { 0,0,0,0 },
} ;

void main( )
{
    Code4 cb ;
    DATA4 *data ;

    //   code4init( &cb ) ;
    cb.safety = 0 ;
    cb.accessMode = OPEN4DENY_RW ;  //must open data file exclusively
                                        //when making a production index with
                                        //Index4::create

    data = d4create( &cb, "PERSON", fields, 0 ) ;

    if( data )
        printf( "PERSON data file successfully created.\n" ) ;

    Tag4info tags( cb ) ;
    tags.add( "NAME_TAG", "NAME" ) ;

    Index4 index ;
    Data4 dataCpp( data ) ;
    index.create( dataCpp, tags ) ;
    if( index.isValid( ) )
        cout << "PERSON index file successfully created." << endl ;

    code4close( &cb ) ;
    code4initUndo( &cb ) ;
}
```

In this particular application, the **Data4** object could have been eliminated since it is never used other than in the **Index4::create**. The **Data4** object could have been constructed within the **Index4::create** with the following line of code:

```
index.create( Data4( data ), tags.tags( ) ) ;
```

or, since there is a **Data4** constructor that takes a **DATA4** structure pointer as a parameter:

```
index.create( data, tags.tags( ) ) ;
```

In the second re-write, the **Data4** constructor is called implicitly by the compiler.

Appendix I: Code Generation

```
GEN4 { [global switches] -d <dataFile> [local switches]
                         [-d <dataFile> [local switches] ] [ ... ] }
     | <inputFile>
```

You can now automatically generate C++ classes corresponding to a specific data file. The generated class defines and initializes all of the **Field4** and **Tag4** objects related to the data file.

Program GEN4 is designed to save you time setting up the basic data base classes used in normal applications. Most application designs require, at some point or another, information from every field of every data base. However, in applications with data files containing many fields, the typing in of all the names can be quite time consuming.

What you need

You need the source code generation program 'GEN4'. This program is installed as source code in the file 'GEN4.CPP' in the EXAMPLES directory.

GEN4.CPP is index specific and it must be compiled as a stand-alone application. CodeBase provides three prebuilt versions of the program, each corresponding to an index file format. Use the GEN4FOX.EXE when you are using FoxPro .CDX index files. If you are using dBASE IV .MDX index files, use the GEN4MDX.EXE and when you are using Clipper .NTX index files, then use the GEN4CLI.EXE.

Client-Server Configuration

Since the executables are stand-alone utilities, they only access local files. These utilities do NOT access the server when they are executed from a client under the client-server configuration.

The GEN4.CPP source code can only be recompiled under the stand-alone configuration of CodeBase to produce new executables. New executables can NOT be built under the client-server configuration.

See 'COMPILER.TXT' for information on how to build GEN4 from scratch.

Running the Generator

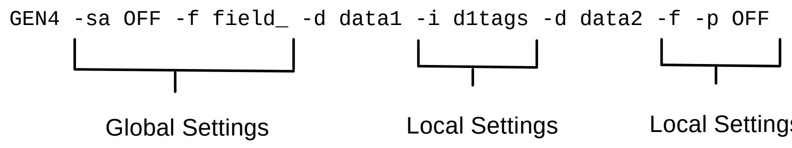
At the most basic level, simply invoke the generator with the name of a data file, and GEN4.EXE generates the class definition using the default values.

```
GEN4 -d datafile.dbf
```

Switches

There are numerous switches which can be used with the GEN4 code generator. Switches placed before the first data file in the list (-data switch) are global and apply to all data files listed. Switches after a data file name are local and apply to that data file only. Local switches do not effect other data files used.

Note If a switch appears in both the global and local settings areas, the local setting is used in the place of the global setting (if possible). This local switch does not effect subsequent data files listed.



```
GEN4 -sa OFF -f field_ -d data1 -i d1tags -d data2 -f -p OFF
```

Figure A.1

GEN4 uses an intelligent algorithm to determine the switch naming convention. That is, if enough characters of a switch are entered to make it unambiguous, GEN4 accepts the switch. For instance, the '-create' switch is distinguished from '-case' after the second character, but may also be entered as '-cre', '-crea', or even '-creat'.

Input File

An input file is an ASCII file which contains GEN4 command line parameters. Using an input file is no different than using command line parameters. However, if an input file is used, the parameters can be placed on one or more lines.

```
GEN4 inputfile
```

A sample input file could look like:

Sample input file

```
-sa OFF -f field_ -o myFile.cpp
-d data1.dbf -t d1tag_
-d data2.dbf -cre OFF
```

Required Switches

SWITCH	SETTING
-data -d	{ *dataFile* } This switch is used to specify the data file name for the code being generated.

Global Only Switches

SWITCH	SETTING
-header -h	{ *headerName* } Specifies the name of the generated header file. If this switch is not provided, the default header name is 'GEN.HPP'.
-output -o	{ *sourceName* } Specifies the name of the generated source file. If this switch is not provided, the default source code file name is 'GEN.CPP'.
-safety -sa	{ **ON** | OFF} Specifies whether safety is on when generating code. When the source and/or header files to be generated already exist and safety is ON (the default), GEN4 aborts the generation and returns an error message. Turn safety to OFF in order to overwrite previously generated code files.

Local Only Switches

SWITCH	SETTING
-index -i	{ *indexFile* } A data file can have several index files. If you want to specify additional index files, use this switch. Code is generated to automatically open/create the index file and to define and initialize tag objects for tags contained in the index file.

Global or Local Switches

SWITCH	SETTING
-case -ca	{ UPPER | **LOWER** | MIXED} By default, defined tag and field objects are in lower case. This switch is used to change this default. You can change it to upper case, or mixed case. With the default, you could potentially get a compile error if a field name is a reserved C++ keyword or is the same as a **Data4** member.
-classname -cl	{ *derivedName* } [*baseName*] By default, the name of the generated class is the same as the data file name (see the case option). In addition, the generated class is derived from **Data4** to provide full data base capabilities. '*derivedName*' may be used to specify the name of the generated class. The optional '*baseName*' is the name of a class to replace '**Data4**' as the base class. Why would you want to replace '**Data4**'? You might want to define a class derived from **Data4** which extents its functionality, and derive the generated class from your own.
-create -cr	{ **ON** | OFF } By default, code is generated to create data and index files if they do not already exist. To generate code only to open the files, specify '-create OFF'.
-fieldid -f	[*fieldIdPrefix*] When a **Field4** object is placed in the generated class, it uses the name of the field as the name of the object. You can add a prefix to the **Field4** objects to distinquish them from **Tag4** objects of the same name (or from C++ reserved words) by specifying this switch. If this switch is used but *fieldIdPrefix* is not provided, the switch is ignored.
-production -p	{ **ON** | OFF } Specifies whether tag objects for production index files should automatically be defined and initialized.
-tagid -t	[*tagIdPrefix*] When a **Tag4** object is placed in the generated class, it uses a default prefix 'tag_' and the name of the tag as the name of the object. You can add a different prefix to the **Tag4** objects, or eliminate it, as long as the objects' names are distinquished them from the **Field4** objects by specifying this switch. If this switch is used and *tagIdPrefix* is not provided, no prefix is used.

Example Switch Settings

Listed below are several switch combinations which are all accepted by GEN4.

```
GEN4 -safety OFF -case MIXED -data datafile.dbf
GEN4 -sa OFF -ca MIXED -d datafile
GEN4 -f FIELD_ -t TAG_ -ca UPPER -d datafile
GEN4 -d data1 -d data2
GEN4 -o classes -d data1 -d data2
GEN4 -d name -i last -i first -d city -i state -i country
```

Example Code

Let's assume that you use the 'DATAFILE.DBF' and the corresponding index file. DATAFILE.DBF has the following fields: FIRST_NAME, LAST_NAME, and AGE (two character fields and a numeric field).

The index file contains a single tag, DATAFILE, which is a compound key built on FIRST_NAME and LAST_NAME.

To generate the class for this data file, the following line may be entered at the command line.

```
GEN4 -d DATAFILE
```

The following code is generated:

```
// GEN.HPP  Generated File - Avoid hand modification.

class DATAFILE : public Data4
{
    public:
        open( Code4&, char * ) ;

        Field4 firstName ;
        Field4 lastName ;
        Field4 age ;
        Tag4 tagDatafile ;
} ;
```

The Header File

The header file is fairly straight forward. It contains a class definition corresponding to the data file. First, by default, the class is named 'DATAFILE' (which is the same as the data file name) and the class is derived from class **Data4**.

Function *open()* is also declared. This function's code is in file 'GEN.CPP' and it will be discussed in depth later.

Finally, **Field4** objects are declared for each field in the data file and for each tag in the data file's production index file.

```
// GEN.CPP  Generated File - Avoid hand modification.
// Created: Jun 23, 1993
```

```
#include "d4all.hpp"
#include "GEN.HPP"

static FIELD4INFO DATAFILEFieldInfo[] =
{
    { "FIRST_NAME", 'C', 20, 0 },
    { "LAST_NAME", 'C', 20, 0 },
    { "AGE", 'C', 3, 0 },
    { 0,0,0,0 }
} ;

static TAG4INFO DATAFILETagInfo[] =
{
    { "DATAFILE", "LAST_NAME+FIRST_NAME", 0, 0, 0 },
    { 0,0,0,0,0 }
} ;

int DATAFILE::open( Code4& cb )
{
    int saveAutoOpen = cb.autoOpen ;
    int saveSafety =  cb.safety ;

    int rcOpenData = 0, rcOpenIndex = 0, rcCreateIndex = 0 ;

    cb.autoOpen = 0 ;

    rcOpenData = Data4::open ( cb, "DATAFILE" ) ;
    if( rcOpenData != 0 )
    {
        cb.safety = 1 ;
        cb.errorCode = 0 ;
        create( cb, "DATAFILE" , DATAFILEFieldInfo ) ;
    }

    cb.safety = 0 ;
    Index4 index ;
    if( rcOpenData == 0 )
        rcOpenIndex = index.open( *this, "DATAFILE" ) ;
    if( isValid() && rcCreateIndex == 0 &&
        (rcOpenData != 0 || rcOpenIndex != 0) )
    {
        cb.errorCode = 0 ;
        rcCreateIndex = i ndex.create( *this, 0, DATAFILETagInfo ) ;
    }

    cb.safety = saveSafety ;
    cb.autoOpen = saveAutoOpen ;

    if( cb.errorCode != 0 )
    {
        if( isValid() )
            close() ;
        return cb.errorCode ;
    }

    firstName.init( *this, "FIRST_NAME" ) ;
    lastName.init( *this, "LAST_NAME" ) ;
    age.init( *this, "AGE" ) ;

    tagDatafile.init( *this, "DATAFILE" ) ;
    return 0 ;
}
```

The Generated Source Code

The generated source code contains the static definition of the data and index file, and the member function *DATAFILE::open*. You will observe that the first part of the generated code for *DATAFILE::open* opens the data and index files. If they are not present, they are created.

This default of creating the files if they are not present can be quite useful for the following reasons: First, if you want to test your program, you can copy it to a new directory and the program will create the data files it needs automatically. Second, if the index files become corrupted, you can just delete them, and they will be recreated. Finally, when you ship your software, you do not need to ship the data and index files with them.

However, the most critical part of the generated code is at the end:

```
firstName.init( *this, "FIRST_NAME" ) ;
lastName.init( *this, "LAST_NAME" ) ;
age.init( *this, "AGE" ) ;
tagDatafile.init( *this, "DATAFILE" ) ;
```

This section of code initializes the field and tag objects declared in the header file.

Using the Generated Code

Once generated, the new class may be used as is, or it may be used as a base class for another class which improves on its functionallity even further.

Listed below is an example program which uses the generated DATAFILE class as a base for another class.

```
#include "d4all.hpp"
#include "gen.hpp"

#ifdef __TURBOC__
    unsigned _stklen = 10000 ; // for all Borland compilers
#endif

class DataFile : public DATAFILE
{
    public:
        sampleAppend( ) ;
} ;

DataFile::sampleAppend( )
{
    appendStart( ) ;

    firstName.assign( "HENRY" ) ;
    lastName.assign( "JONES" ) ;
    age.assignLong( 37 ) ;

    return append( ) ;
}

void main( )
{
    Code4 cb ;
    DataFile data ;

    int rc = data.open( cb, "datafile" ) ;
    if( rc < 0 ) return ;

    data.sampleAppend() ;
    cb.closeAll( ) ;
}
```

First, this example derives class 'DataFile' from the generated class *DATAFILE*. The idea is to have a user defined class to create functions which operate on the specific data file object. Class *DATAFILE* could be modified to declare the additional function. However, we want to avoid modifying the generated code, since the changes would be lost if the code was regenerated.

```
class DataFile : public DATAFILE
{
    public:
        sampleAppend( ) ;
} ;
```

DataFile::sampleAppend calls **Data4::appendStart**, assigns values to its field objects and then calls **Data4::append** to append the record.

Since **Data4** is a base class of *DataFile*, all of the CodeBase **Data4** member functions can be referenced directly from within *DataFile* member functions. Similarly, all of the field and tag objects, declared in the generated base class *DATAFILE*, can also be referenced directly.

```
firstName.assign( "HENRY" ) ;
```

The above example illustrates how you can put the object oriented capabilities of C++ and CodeBase to work when using the generated code.

Index