Clarion Language Programming Guide

SoftVelocity

SoftVelocity Incorporated
2769 East Atlantic Blvd.
Pompano Beach, Florida 33062
(954) 785-4555
www.softvelocity.com

Printed in the United States of America (0703)

Contents:

1 - Program Structure 7

Structured Programming 7
PROCEDUREs 7
Local Stack-Based Data Declarations 7
PROGRAM MAP 8
MODULE 8
MEMBER 8
MEMBER MAPs 9
MODULEs within MEMBER MAPs 10
PROCEDURE MAPs 12
Summary 13

2 - Easing Into OOP 15

Opening Shots 15
Procedural Code Re-Visited 15
CLASS Declaration 16
Encapsulation 16
Instantiation 17
Field Qualification Syntax 19
Constructors and Destructors 19
More About Encapsulation 21
Inheritance 22
As Easy As Apple Pie 23
Multiple Inheritance 23
Composition 24
And Even More About Encapsulation 25
And More On Constructors and Destructors 26
PARENT 27
Polymorphism 27
Virtual Methods 28
More Apple Pie 29
Late Binding 30
Local Derived Methods 31
Summary 31

3 - Object Oriented Programming (OOP) 33

Object Overview - What are Objects? 33
Why Objects? 33
What Makes an Object? 34
Clarion's OOP Extensions 36
The CLASS Structure—Encapsulation 36
CLASS Properties 36
CLASS Methods 36
Creating Objects 37
Using References as Properties 38

Constructors and Destructors 39
Public vs. PRIVATE vs. PROTECTED....40
Derived CLASSes—Inheritance....42
Overriding Inherited Methods....44
Multiple Inheritance vs. Composition 47
Virtual Methods—Polymorphism....49
Local Derived Methods 51
Summary....52

4 - Database Design 53

Database Design....53
Relational Database Design....53
File Relationships....55
Translating the Theory to Clarion....57
Referential Integrity....58
Nullify the Foreign Key....62
Summary....64

5 - Data File Processing.... 65

Data File Processing....65
File Access Methods....65
KEY and INDEX....65
Sequential File Access....66
Random File Access....69
Summary....71

6 - Multi-User Considerations 73

Multi-User Considerations....73
Opening Files....73
Concurrency Checking....74
HOLD and RELEASE....79
LOCK and UNLOCK....82
“Deadly Embrace”....84
Summary....85

7 - Developing Client/Server Applications 87

Introduction to Client/Server.... 87
Client/Server Defined....87
Types of Client/Server Database Applications....87
SQL Database Engines....88
Clarion and SQL....88
Database Design and Network Traffic.... 89
Referential Integrity Handling....89
Data Validation....90
Clarion Language Client/Server Support.... 90
The VIEW Structure....90

The BUFFER Statement 92
Embedded SQL in Clarion 93
NULL Data Handling 95
Error Handling 96

Index: 97

1 - Program Structure

Structured Programming

The “proper” structure of a computer program is a topic that can be the beginning of a highly charged debate. Many programmers have definite, strongly held, ideas about what constitutes “proper” structure for a program, and those ideas do not always conform to another programmer’s thoughts concerning “proper” structure. Therefore, this essay is a general discussion of the Clarion Language tools which provide the ability to construct programs in your own concept of “proper” structure.

PROCEDUREs

The key to any structured programming is the ability to break your program code into separate tasks to call when needed. The Clarion Language provides a statement which allows this type of task separation: PROCEDURE. A PROCEDURE can be prototyped to RETURN a value, and therefore may be called as part of an expression or parameter list. A PROCEDURE which does not RETURN a value may only be called explicitly as a separate program statement—it may not be used as part of an expression or parameter list.

A PROCEDURE prototyped to RETURN a value may also be called explicitly as a separate program statement when the returned value is unimportant to the context in which it is called. Doing this will generate warnings from the compiler (which may be safely ignored) unless the PROCEDURE’s prototype has the PROC attribute.

Within a procedure you may put repetitive executable code statements into ROUTINEs. This is useful for optimizing the size of your program code and moving explicit functionality out of the main procedure logic, making the overall logic flow easier to follow. However, ROUTINEs are only local to the procedure and may only be used within the procedure in which it resides. Therefore, the use of ROUTINEs is not part of this discussion.

Local Stack-Based Data Declarations

Every procedure has a data declaration section and an executable code section. The data declaration section follows the keyword PROCEDURE and ends with the keyword CODE. The executable code section follows the keyword CODE. Any variables or data structures declared in a procedure*’s* data declaration section are local to that procedure. This means they are only visible within that procedure, unless passed as a parameter to another procedure.

Variables declared locally in a procedure (without the STATIC attribute) are dynamically allocated memory when the procedure executes. The memory for a local viable is allocated on the program’s stack unless it is larger than the stack threshold, then it is allocated on the heap (but it behaves the same as a variable allocated on the stack). When the procedure is complete and program control returns to the place from which it was called, the local variable’s memory is de-allocated and returned to the program for other uses. Only variables declared local to a procedure are dynamically allocated memory on the stack.

Dynamically allocated local variables makes recursive and reentrant procedures possible. A procedure is recursive if it calls itself within its code. Each time a procedure is recursively called, it receives a new copy of its non-static local variables. Recursion is an advanced programming technique which is useful for procedures which must be executed in successive iterations.

PROGRAM MAP

Just like a procedure, a PROGRAM has a data declaration section (between the keywords PROGRAM and CODE) and an executable code section (following the keyword CODE). All variables and data structures declared in the PROGRAM's data declaration section are global (available for use anywhere in the application) and are allocated memory statically.

A program's MAP structure is located in the global data declaration section. This MAP tells the program what procedures are globally available anywhere in the application.

In a Clarion program, all procedures must be prototyped in a MAP structure, unless they are methods of a CLASS (then the CLASS structure itself contains the prototype). A prototype of a procedure tells the compiler the name of the procedure, and how to deal with it. For a complete description of prototypes, see PROCEDURE *Prototypes* in the *Language Reference*.

MODULE

Within a MAP, you may have MODULE structures which declare the separate source file containing the procedures prototyped in that MODULE structure. The MODULE structure is the mechanism which allows the next level of program organization: grouping procedures into separate source files.

There are many reasons to split off groups of procedures into separate source files. This is the point at which debate amongst programmers with differing viewpoints becomes highly charged. Grouping procedures which accomplish related tasks is one approach. Grouping for optimization of compile times is another reason for grouping certain files together. Another purpose of procedure grouping is the eventual creation of a Dynamic Link Library (DLL). No matter the reason, the MODULE structure defines the method by which your program may reflect your structural ideas.

MEMBER

The prototypes for procedures which are defined in a source file other than the PROGRAM file must be declared in a MODULE structure within a MAP. The source code file specified by the MODULE structure must begin with a MEMBER statement. Any source code file beginning with a MEMBER statement is commonly referred to as a " MEMBER module."

The MEMBER statement specifies the filename of the PROGRAM source file to which the MEMBER file belongs. The MODULE structure in the MAP points to the MEMBER source file, and the MEMBER statement within that source file points back to the PROGRAM source file.

For example, a source code file named MYPROG.CLW contains the following code:

```
PROGRAM                          !Begin global data declaration section

 MAP                             !The global MAP structure
Proc1  PROCEDURE                 !A prototype for a procedure whose
                                 !source code is in MYPROG.CLW
  MODULE('MYPROG2')              !A separate source file, MYPROG2.CLW
Proc2  PROCEDURE                 !  contains another procedure
  END                            !End MODULE
 END                             !End MAP

 CODE                            !Begin program executable code
   !some executable source code

Proc1 PROCEDURE                  !Begin local data declaration section
 CODE                            !Begin procedure executable code
                                 !some executable source code
```

A second source code file, MYPROG2.CLW, declared in the MODULE structure of the PROGRAM MAP in the example code above, contains the following code:

```
MEMBER('MYPROG')                !MEMBER module of the PROGRAM in MYPROG.CLW

Proc2 PROCEDURE                 !Begin local data declaration section
 CODE                           !Begin procedure executable code
   !some executable source code
```

In this example, the program has two procedures, Proc1 and Proc2. The source code for Proc2 is in a separate file, MYPROG2.CLW. Therefore, the PROGRAM MAP contains a MODULE structure which declares that Proc2 is in source file MYPROG2.CLW, and the first statement in MYPROG2.CLW is a MEMBER statement which tells the compiler the PROGRAM file to which this MEMBER belongs.

MEMBER MAPs

A MEMBER module also has a data declaration section. It begins following the keyword MEMBER and ends at the first PROCEDURE statement in the module.

Any variable or data structure declared in the data declaration section of a MEMBER module is local to that MEMBER module and is allocated static memory. This means it may be accessed only by a procedure which resides in that MEMBER module, unless passed as a parameter to another procedure which resides in another source module (essentially making them “private” variables).

In addition to data declarations, a MEMBER module’s data declaration section may also contain its own MAP structure. This MAP is structurally the same as the MAP in the PROGRAM module, and declares procedure prototypes which are available locally within the MEMBER module.

Using the previous example, a file named MYPROG.CLW contains the following code:

```
 PROGRAM              !Begin global data declaration section

 MAP                  !The global MAP structure
   MODULE('MYPROG2')  !A separate source file, MYPROG2.CLW
Proc1  PROCEDURE      !  contains a procedure
  END                 !End MODULE
 END                  !End MAP

 CODE                 !Begin program executable code
   !some executable source code
```

The second source code file, MYPROG2.CLW, now contains the following code:

```
 MEMBER('MYPROG')    !The beginning of a MEMBER module's data
                     !declaration section
 MAP                 !The MEMBER MAP structure
Proc2  PROCEDURE     !A prototype for a procedure which is
                     !local to the MYPROG2.CLW MEMBER module
 END                 !End MAP
Var1 BYTE            !A static variable local to the MEMBER

Proc1  PROCEDURE     !Begin local data declaration section
 CODE                !Begin procedure executable code
   !some executable source code

Proc2  PROCEDURE     !Begin local data declaration section
 CODE                !Begin procedure executable code
   !some executable source code
```

The Proc2 procedure was not declared in the PROGRAM MAP, but only in the MEMBER module's MAP. Therefore, it may be called only by other procedures within the MYPROG2.CLW MEMBER module (Proc1). Var1 was declared in the MEMBER data declaration section, therefore it may be used only by the procedures actually residing in the MEMBER module—Proc1 and Proc2.

MODULEs within MEMBER MAPs

Just as in the PROGRAM MAP, a MEMBER module's MAP may also contain MODULE structures, if the procedures prototyped are in separate source files. Any procedure which is not prototyped in the PROGRAM MAP must be prototyped in the MEMBER module MAP in which it resides. This means that identical procedure prototypes are required:

In the MODULE structure of the MEMBER module MAP in the source code file which calls the procedure.

In the MAP structure of the second MEMBER module, which actually contains the source code definition of the procedure.

Again using the previous examples, a file named MYPROG.CLW contains the following code:

```
 PROGRAM                !Begin global data declaration section
 MAP                    !The global MAP structure
   MODULE('MYPROG2')    !A separate source file, MYPROG2.CLW
Proc1  PROCEDURE        !contains a procedure
   END                  !End MODULE
 END                    !End MAP
 CODE                   !Begin program executable code
   !some executable source code
```

The second source code file, MYPROG2.CLW, now contains the following code:

```
  MEMBER('MYPROG')     !The beginning of a MEMBER module's data
                       !  declaration section
 MAP                   !The MEMBER MAP structure
   MODULE('MYPROG3')   !A separate source file, MYPROG3.CLW
Proc2  PROCEDURE       !contains another procedure
   END                 !End MODULE
 END                   !End MAP

Proc1  PROCEDURE       !Begin local data declaration section
 CODE                  !Begin procedure executable code
 Proc2                 !A call to Proc2
  !some executable source code
```

A third source code file, MYPROG3.CLW, contains the following code:

```
 MEMBER('MYPROG')     !The beginning of a MEMBER module's data
                      !  declaration section
 MAP                  !The MEMBER MAP structure
Proc2  PROCEDURE      !An identical procedure prototype to the
                      !  prototype declared in the
                      !  MYPROG2.CLW MEMBER module MAP
 END                  !End MAP

Proc2  PROCEDURE      !Begin local data declaration section
 CODE                 !Begin procedure executable code
  !some executable source code
```

The procedure prototype for Proc2 in the MODULE structure of the MAP in MYPROG2.CLW is duplicated in the MAP in MYPROG3.CLW. This duplication is required for all procedures declared in a MODULE structure of a MEMBER module's MAP.

The procedure prototype for Proc2 may be placed in as many MEMBER module MAP structures (within its MODULE structure) as are needed to allow as many procedures to call it as need to. This allows you to explicitly declare and use Proc2 only in the modules that need it, and in as many modules as actually do need it.

There is an advantage to explicitly declaring all procedures in the PROGRAM in local MEMBER MAPs instead of a single global MAP. That advantage is compile time—whenever you add a procedure to a global MAP, you change the PROGRAM module, which forces a re-compilation of the entire application. By explicitly prototyping all the application's procedures in MEMBER MAPs only in the modules in which the procedures are used, you can eliminate many global re-compilations, saving you time as you build the application.

PROCEDURE MAPs

You recall that a PROCEDURE has a data declaration section, following the keyword PROCEDURE and ending at the CODE statement. In addition to data declarations, a PROCEDURE's data declaration section may also contain its own MAP structure. A PROCEDURE's MAP may not contain any MODULE structures, and declares procedure prototypes which are available locally within the PROCEDURE. These are called Local Procedures.

The definitions of any Local Procedures must immediately follow the PROCEDURE within which they are declared (in the same source code module). The advantage of Local Procedures is that they share the local data and routines of their declaring PROCEDURE. Conceptually, a Local Procedure could be thought of as a routine which can receive parameters, return a value, and contain its own data declarations.

Using the previous example, a file named MYPROG.CLW contains the following code:

```
Proc1  PROCEDURE          !Begin Local data declaration section

 MAP                      !The local MAP structure
Proc2  PROCEDURE          !  contains a local procedure declaration
 END                      !End MAP

ProcLocalVariable LONG    !A local variable declaration

 CODE                     !Begin procedure executable code
 DO MyRoutine             !Call a ROUTINE

MyRoutine ROUTINE
 ProcLocalVariable += 1   !Increment the local variable


Proc2  PROCEDURE          !Local Procedure definition
                          ! must follow the declaring PROCEDURE in the
                          ! same source module
LocalVariable LONG        !A local variable declaration
 CODE
 LocalVariable = ProcLocalVariable
                          !Use a variable from the declaring PROCEDURE
 DO MyRoutine             !Call a ROUTINE from the declaring PROCEDURE
```

The Proc2 procedure was not declared in the PROGRAM MAP or a MEMBER MAP, but only in the Proc1 PROCEDURE MAP. Therefore, it may be called only by the Proc1 procedure. ProcLocalVariable was declared in the Proc1 PROCEDURE's data declaration section, therefore it may be used only by the declaring procedure and any Local Procedures—Proc1 and Proc2. MyRoutine ROUTINE is also available only to the declaring procedure and any Local Procedures—Proc1 and Proc2.

Summary

- With the Clarion Language you may separate tasks into procedures.
- A procedure may contain ROUTINEs which optimize repetitive source code.
- A procedure may have local stack-based variables which optimize run-time memory requirements and make recursion possible.
- The PROGRAM data declaration section declares all data which is globally available.
- The PROGRAM MAP declares the prototypes for all procedures which are globally available.
- A MAP may contain a MODULE structure which points to the source code for procedures which reside in separate source files.
- The MEMBER statement points back to the PROGRAM source file.
- The MEMBER module data declaration section may declare variables and data structures which are only available to the procedures that reside in the MEMBER module.
- A MEMBER module may have its own MAP structure declaring other procedures only known locally in the MEMBER module.
- A MEMBER module's MAP may contain MODULE structures which point to procedures that reside in other MEMBER modules. The MEMBER which the MODULE points to must also contain a MAP with identical prototypes for those procedures.
- Explicitly prototyping all of an application's procedures in MEMBER module MAP structures only where actually needed can significantly reduce the number of global re-compilations of the application required during development.
- Local Procedures have the advantage of sharing their declaring procedure's local data and routines. Conceptually, this makes them function like routines which can declare data, receive parameters, and return values.

2 - Easing Into OOP

This chapter was derived from a speech delivered by Richard Taylor at the 1997 Developer's Conference (DevCon '97) prior to the release of Clarion Version 4. Several attendees suggested that it become part of the Clarion documentation set.

Opening Shots

I know that some of you have had a lot of experience with Object Oriented Programming in other languages, while others here have never even *said* "OOPs" in their lives.

For those of you who do have a lot of OOP background, please bear with me because I'm going to be explaining some things that may seem pretty obvious to you. I apologize in advance if things seem a little simplistic.

To the rest of you: this will be a fairly quick overview. There are many books that have been written on Object Oriented Programming, so I only intend to hit these concepts "once over lightly" in relation to how we've implemented them in the Clarion language. OK, let's start from a point that we all should be familiar with — Procedural code.

Procedural Code Re-Visited

We all know that a PROCEDURE has a Data Section and a Code Section.

And we all know that Local Data variables declared in a PROCEDURE are only visible within the PROCEDURE and exist only while the PROCEDURE executes. Therefore, Local Data variables are in scope only within the PROCEDURE in which they are declared.

When the PROCEDURE is called, the Local variables are automatically allocated memory on the stack. This makes them available for use in the PROCEDURE's executable code statements.

Then, when a RETURN statement executes in the PROCEDURE to return to the place from which it was called, the memory used by the Local variables is automatically de-allocated from the stack and returned to the operating system.

So we can say that the lifetime and visibility of local data in a PROCEDURE is limited to the duration of the PROCEDURE execution.

OK, so what if you wanted to have multiple CODE sections able to operate on the same set of data variables? In Clarion's Procedural code you could do it two ways:

• make the variables Global,

• or move them into a Module's Data section.

Since Module Data is a concept that's fairly unique to Clarion, we'll only discuss the Global approach this morning.

By declaring your variables Globally you end up with effectively what you want: a single set of Data variables and multiple executable CODE sections that can operate on that data. One drawback though, is that the Global Data variables are not limited in lifetime and there are various schools of programming thought that hold that the proliferation of Global Data in your programs is a bad thing.

Global data is allocated static memory. This means the variables are allocated memory when the program starts execution and only de-allocated when the program terminates. By the way, the same thing is true of Module Data, so you wouldn't buy anything by going that route except limiting the visibility to the PROCEDUREs within the same module.

OK, so you can have a single set of variables that can be referenced by multiple executable CODE sections by declaring the variables as Global Data.

What if you wanted to have multiple *sets* of this data that could all be referenced by the same set of multiple executable CODE sections? In other words, what if you wanted to *re-use* the same executable code over and over again, but using different sets of data variables each time?

Well, you *could* place all the Global data variables inside a QUEUE structure, then somehow manipulate the QUEUE entries so that you could in some way sensibly maintain which QUEUE entry it is that you're currently working on. However, this would get pretty complex pretty quickly.

There is a better way. And that way is the CLASS declaration.

CLASS Declaration

The Clarion CLASS structure allows you to declare data variables — both simple data types and reference variables — and the PROCEDUREs which operate on them. This is the basic declaration structure of Object Oriented Programming in Clarion. This is the structure that lets you re-use code over and over again for different sets of data.

```
MyClass      CLASS
Property      LONG
QueProperty   &SomeQueue
Method1       PROCEDURE
Method2       PROCEDURE(LONG),LONG
              END
```

The CLASS structure is a container which holds a set of data variables and the PROCEDUREs which operate on them. To use the industry-standard terms for these, the CLASS structure contains properties and methods. Properties are the data, and methods are the PROCEDUREs which act on those properties. Taken together, these properties and methods form a single Object.

OK, so now we've introduced and defined in Clarion terms three standard OOP buzzwords: Properties, Methods, and Object. Let me introduce the first of the three major OOP buzzwords: Encapsulation.

Encapsulation

The CLASS structure is a container for the properties and methods. According to the Oxford English Dictionary, the word Encapsulate means: *to enclose (as) in a capsule*. Therefore, the CLASS structure *encapsulates* the properties and methods declared within it.

Now, you OOP purists out there are about to say, "Wait a minute, it means a lot more than that in OOP." And you're right. In common usage for OOP, encapsulation means much more than just containerizing the properties and methods, and I'll get around to all that shortly. Just be patient.

OK, so we've now got a container for properties and methods. What good does that do us? Well, you'll notice that the Clarion keyword we're using here is CLASS and not OBJECT. The reason for that is because a CLASS declaration declares a specific type of object, and it can also declare an instance of that object type (but it doesn't have to). So here's the next ten-dollar OOP buzzword we need to learn: Instantiation.

My Oxford English Dictionary defines Instantiate as: *To represent by an instance*.

It also defines Instantiation as: *The action or fact of instantiating; representation by an instance*.

The real power of the CLASS structure is that you can have multiple *instances* of the type of object the CLASS declares. Each of these instances is allocated its own set of properties, but they all share the same set of methods that operate on those properties. That's where the real promise of OOP lies, in the fact that you only need a single set of methods to operate on the properties of any object instance of a particular CLASS. The memory for the properties of a specific object instance is allocated when that object is instantiated (a ten dollar word for created).

OK, so how do we instantiate an object? There are three ways to do it:

1. A CLASS declared without the TYPE attribute declares both a type of object and an instance of that object type.
2. A simple declaration statement with the data type being the label of a previously declared CLASS structure declares an object of the same type as the CLASS.
3. Declare a reference to the label of a previously declared CLASS structure, then use the NEW and DISPOSE procedures to instantiate and destroy the object.

That's how you instantiate objects. Now you might ask, "Why is he making such a big deal about instantiation?" Good question.

The answer has to do with memory allocation for the properties that each object contains. Remember that each object gets its own discrete set of properties when the object is instantiated, so it's important to know when these things are instantiated.

Instantiation

An object declared in your program's Global data section is instantiated for you automatically at the CODE statement which marks the beginning of the Global executable code. The object is automatically destroyed for you when you RETURN from your program to the operating system. Therefore, the lifetime and visibility of a Global object is Global.

```
PROGRAM                   !Global Data and Code
MyClass  CLASS            !Declare an object and
Property  LONG            !a type of object
Method    PROCEDURE
         END
ClassA MyClass            !Declare MyClass object
ClassB &MyClass           !Declare MyClass reference
 CODE                     !MyClass and ClassA automagically instantiated
 ClassB &= NEW(MyClass)   !Explicitly Instantiate object
  ! execute some code
 DISPOSE(ClassB)          !Destroy object (required)
 RETURN                   !MyClass and ClassA automagically destroyed
```

Objects declared in any module's data section are also instantiated for you automatically at the CODE statement which marks the beginning of the Global executable code and automatically destroyed for you when you RETURN from your program to the operating system. This makes their lifetime Global, despite the fact that they are only visible to PROCEDUREs within the same module.

```
 MEMBER('MyApp')     !Module Data

MyClass   CLASS      !Declare an object and
Property   LONG      ! a type of object
Method     PROCEDURE
          END
ClassA   MyClass     !Declare MyClass object
ClassB   &MyClass    !Declare MyClass reference

SomeProc  PROCEDURE
 !MyClass and ClassA are instantiated and
 !  destroyed at the same time as the
 !  Global objects
 !ClassB must be explicitly instantiated
 !  and destroyed in some PROCEDURE in
 !  the module
```

Objects declared in a PROCEDURE's data section are instantiated for you automatically at the CODE statement which marks the beginning of the PROCEDURE's executable code, and automatically destroyed for you when you RETURN from the PROCEDURE. This limits their lifetime and visibility to the PROCEDURE within which they are declared.

```
SomeProc  PROCEDURE   !Local Data and Code
MyClass   CLASS       !Declare an object and
Property   LONG       ! a type of object
Method     PROCEDURE
          END
ClassA   MyClass      !Declare MyClass object
ClassB   &MyClass     !Declare MyClass reference
 CODE                 !MyClass and ClassA automagically instantiated
 ClassB &= NEW(MyClass)  !Instantiate object
  ! execute some code
 DISPOSE(ClassB)      !Destroy object (required)
 RETURN               !MyClass and ClassA automagically destroyed
```

OK, all that's pretty straight forward – about what you expected, right? The ones you have to keep your eye on are the objects that you declare as reference variables. These objects are only instantiated when you execute a NEW statement, and – now here's the thing to watch out for – they are never automatically destroyed for you, you must use DISPOSE to get rid of them yourself. If you forget to do that, you can end up with a memory leak in your program.

So the question now comes up, "If each object gets its own set of properties but shares the same set of methods, how does the executable code inside the method know exactly which object's properties to affect?" Another good question.

The beginning of the answer lies in Clarion's Field Qualification syntax.

Field Qualification Syntax

The properties and methods of an object are referenced in code by prepending the name of the object to the property or method using what's come to be known as "dot syntax." This is pretty standard syntax in all the various languages that support OOP. For example, assume you have an object named "Fred" and it has a method named "Work" and two properties named "Pay" and "Hours." Outside the object itself you might call that method in an assignment as "Fred.Pay = Fred.Work(Fred.Hours)"

Now, within the methods of an object, there is no way to know exactly what the object is that is currently executing. Remember, you can have many separate instances of a specific type of object which all share the same set of methods. Therefore, within the methods of an object, instead of using the object's name, you use the word SELF.

That changes the previous assignment, when made within a method of an object, to "SELF.Pay = SELF.Work(SELF.Hours)" because the code at that point doesn't know whether it is referring to the properties of the Fred object or the Barney object, or the Bruce or Richard objects. It doesn't know and doesn't care, because SELF always means "whatever the current object is."

```
 PROGRAM
Employee  CLASS,TYPE        !Declare object TYPE
Pay        DECIMAL(7,2)
Hours      DECIMAL(3,1)
CalcPay    PROCEDURE
Work       PROCEDURE(*DECIMAL),DECIMAL
          END
Fred    Employee         !Declare object instances
Barney  Employee
 CODE
 Fred.Pay = Fred.Work(Fred.Hours)   !Code outside the "Fred" object

Employee.CalcPay  PROCEDURE        !Method Definition
 CODE
 SELF.Pay = SELF.Work(SELF.Hours)   !Code inside object method
```

So the answer to the question, "How does the code inside the method know exactly which object's properties to affect?" is the use of SELF as the object name within the method's code.

Constructors and Destructors

There are two more things to know about what happens at an object's Instantiation and destruction. Those two things add two more OOP buzzwords to our collection: Constructors and Destructors.

When you talk about Constructors and Destructors in other OOP languages what you're actually talking about are Automatic Constructors and Destructors. Clarion fully supports Automatic Constructors and Destructors.

An Automatic Constructor is a specific method that is automatically called for you when an object is instantiated, no matter how that object is instantiated (either declared in a data section or explicitly instantiated with the NEW procedure).

To declare an Automatic Constructor in Clarion, all you need to do is create a method named CONSTRUCT in your CLASS. The CONSTRUCT method cannot receive any parameters and cannot return a value.

The most typical usage of Automatic Constructors in other languages is to allocate memory for properties and initialize them. In Clarion, since memory is automatically allocated for variables, the most common usage would probably be to initialize properties to specific starting values and to initialize reference variable properties.

An Automatic Destructor is a specific method that is automatically called for you when an object is destroyed, no matter how that object is destroyed (either by RETURN from the limit of its local scope or explicitly destroyed with the DISPOSE procedure).

Similar to the Constructor, you declare an Automatic Destructor as a method named DESTRUCT in your CLASS. The DESTRUCT method also cannot receive any parameters and cannot return a value.

The most typical usage of Automatic Destructors in other languages is to de-allocate the memory used for properties. In Clarion, since automatically allocated memory is also automatically de-allocated, the most common usage would probably be to DISPOSE of any reference variable properties that need destruction.

```
MyClass   CLASS
Property   LONG
Method     PROCEDURE
Construct  PROCEDURE      !Automatic Constructor
Destruct   PROCEDURE      !Automatic Destructor
          END
```

There is more about Automatic Constructors and Destructors, and is discussed a little later.

More About Encapsulation

To many OOP purists, encapsulation is all about *hiding* the properties and methods of the CLASS from the rest of the program, not just containerizing them. This makes them “Private Property” and the Clarion language fully supports this with the PRIVATE attribute.

The PRIVATE attribute on a property declaration or a method prototype means that the property or method is visible only to the other methods within that CLASS. This “hides” them from everything outside the CLASS. Without the PRIVATE attribute, the properties and methods are “public” and the rest of your program can access them whenever the object has been instantiated and is in scope.

```
 PROGRAM
MyClass    CLASS
MyProperty  LONG,PRIVATE        !Private Property
Method      PROCEDURE
MyMethod    PROCEDURE,PRIVATE   !Private Method
           END
 CODE
 MyClass.MyMethod               !Invalid here
 MyClass.Method                 !Valid here
 MyClass.MyProperty = 10        !Invalid here

MyClass.Method PROCEDURE
 CODE
 SELF.MyMethod                  !Valid here
 SELF.MyProperty = 10           !Valid here
```

There’s even more to say about Encapsulation and “public” versus PRIVATE stuff in Clarion, but first we need to introduce another OOP concept.

Inheritance

OK, so far we've learned some major OOP buzzwords: Properties, Methods, Objects, Encapsulation, Instantiation, Constructors and Destructors.

The next one is *Inheritance*. You all know what Inheritance is, don't you? Inheritance is where someone dies and leaves you a million dollars, right? Well, almost.

Inheritance is closely coupled to another common OOP term: to derive. My trusty Oxford English Dictionary defines derive as: *To convey from one (treated as a source) to another, as by transmission, descent, etc.; to transmit, impart, communicate, pass on, hand on.*

To derive a new CLASS, you simply name the parent CLASS as the parameter to the new CLASS statement and the new CLASS declaration starts out with everything the parent CLASS had. The difference here is that, in OOP no one has to die for the child to inherit.

This brings up a couple more common OOP terms: *Base CLASS* and *Derived CLASSes*. A Base CLASS is a CLASS which has no parameter to its CLASS statement — meaning it doesn't inherit anything — while a Derived CLASS always has a parameter naming its Parent CLASS. Notice that I did not say that the parameter to the Derived CLASS statement names a Base CLASS – it doesn't. The parameter to a Derived CLASS statement names its Parent CLASS, which could be a either Base CLASS or another Derived CLASS. This means that you can have multiple generations of inheritance.

```
 PROGRAM
MyClass   CLASS          !Declare Base Class
Property   LONG
MyProperty LONG,PRIVATE !Private = no access in derived class
Method     PROCEDURE
          END
ClassA    CLASS(MyClass)!Declare Derived Class
Aproperty  LONG          ! which inherits
Amethod    PROCEDURE     ! MyClass.Property and
          END            ! MyClass.Method
 CODE
 ClassA.Method           !Valid method call
 ClassA.MyProperty = 10 !Invalid, not accessible outside MyClass
 MyClass.Amethod         !Invalid, inheritance only is one way
```

So, back to Inheritance. A Derived CLASS inherits all the public properties and methods of its Parent CLASS. It also inherits PRIVATE properties and methods, but cannot access them in any of the Derived CLASS methods because they are truly the "Private Property" of the CLASS which declares them.

What this all means is that, when you derive a CLASS from a Parent CLASS, there is no need to duplicate all the functionality already in the Parent CLASS because the Derived CLASS inherits it all.

So why derive at all? So you can re-use your code and maintain it in only one place. Because, when properly designed, a set of CLASSes has all the most common properties and methods for objects of the same or very similar type declared in the Base CLASS. Derived CLASSes then only need to declare the properties and methods which differentiate them from any other CLASSes that might be derived from the same Parent.

As Easy As Apple Pie

One example of this type of design would be designing CLASSes for Apple Pies. The Base CLASS would contain all the standard properties common to all Apple Pies, such as Apple and Crust properties and a Bake method. However, for the individual types of Apple Pies the Derived CLASS would contain the differences between the various varieties of Apple Pies.

For example, the *Dutch ApplePie* Derived CLASS might contain a CrumbleTop property, the Traditional *American ApplePie* Derived CLASS might contain a TopCrust property, and the *Grandmas ApplePie* Derived Class might contain a CaramelTop property.

```
 PROGRAM


ApplePie  CLASS ,TYPE       !Declare Base Class
Apple      STRING(20)
Crust      STRING(20)
Bake       PROCEDURE
          END

Dutch     CLASS(ApplePie)   !Declare Derived Class
CrumbleTop STRING(20)
          END

American  CLASS(ApplePie)   !Declare Derived Class
TopCrust   STRING(20)
          END

Grandmas  CLASS(ApplePie)   !Declare Derived Class
CaramelTop STRING(20)
Bake       PROCEDURE        !Overridden method
          END
```

One other thing that you can do using Inheritance is to override the methods. By giving a method in a Derived CLASS exactly the same prototype (name, parameter list, and any return data type) as a Parent CLASS method, you override the method in the Derived CLASS. This allows you to provide slightly different functionality in a Derived CLASS if you need to.

Multiple Inheritance

You will notice that Clarion's Inheritance syntax only allows you to derive from a single Parent. This is called Single Inheritance. There is such a thing as Multiple Inheritance in OOP theory, but you can thank us for not giving it to you.

By not giving you Multiple Inheritance, there's another ten-dollar buzzword that you DO NOT have to learn anything about – Disambiguate! Since you don't have to know anything about it, I'm not going to define it.

For those cases where multiple inheritance would be useful, there is an OOP technique called Composition which gives you the benefits of multiple inheritance without requiring you to Disambiguate anything!

Composition

The Composition technique simply means inheriting from one CLASS and also declaring a reference to another object as a property of your Derived CLASS. Therefore, your derived CLASS inherits the properties and methods from its Parent CLASS, and also contains a reference to the second object (and implicitly, all its properties and methods) instead of inheriting them. This gives the Derived CLASS full and complete access to all the inherited properties and methods and all the properties and methods of the referenced object it contains. The ABC Library classes use Composition in several places, where it's appropriate to them.

```
 PROGRAM


ApplePie  CLASS ,TYPE     !Declare Base Class
Apples     STRING(20)
Crust      STRING(20)
Bake       PROCEDURE
          END

IceCream  CLASS,TYPE      !Declare Base Class
Flavor     STRING(20)
Scoop      PROCEDURE
          END

AlaMode   CLASS(ApplePie) !Composition: Derive from one CLASS
OnTheSide  &IceCream      ! and contain a reference to
Serve      PROCEDURE      ! an object of another CLASS
          END
```

And Even More About Encapsulation

In addition to “public” and PRIVATE properties and methods, there are also PROTECTED properties and methods. You recall that the PRIVATE attribute means that the property or method is only visible within the methods of the CLASS in which the PRIVATE property or method is declared, and is not accessible by any Derived CLASS. And without the PRIVATE attribute the properties and methods are “public” and visible and available to any code, whether that code is in a CLASS method or not. Well sometimes you don’t want either private or public.

Sometimes you only want a property or method to be available within the CLASS in which it is declared, or any CLASS derived from it. That’s what the PROTECTED attribute is for.

```
 PROGRAM

MyClass   CLASS              !Declare Base Class
Property    LONG
MyProperty  LONG,PROTECTED   !Semi-Private
Method      PROCEDURE
          END

ClassA    CLASS(MyClass)     !Declare Derived Class
Aproperty  LONG
Amethod    PROCEDURE
          END
 CODE
 ClassA.MyProperty = 10      !Invalid out of method

ClassA.Amethod  PROCEDURE
 CODE
 SELF.MyProperty = 1         !Valid within method
```

PRIVATE properties and methods cannot be accessed by Derived CLASSes, but PROTECTED properties and methods can be. PROTECTED properties and methods are not visible outside the CLASS within which they are declared or any CLASS derived from that CLASS. I’d guess this makes the PROTECTED properties and methods sort of “semi-private.”

And More On Constructors and Destructors

And while we're going backwards, remember a while back I said there was more to tell you about Automatic Constructors and Destructors? Well now's a good time to tell you about Inheritance and how it affects Automatic Constructors and Destructors.

What happens when a Parent CLASS has a CONSTRUCT method and the Derived CLASS also needs one? We already talked about overriding methods in Derived CLASSes, so you might guess that the Derived CLASS CONSTRUCT method would simply override the Parent CLASS method. That guess would be wrong, of course.

So why is it wrong? Because overriding the Parent CLASS Constructor might mean that code that's required to initialize inherited properties would not execute. If the inherited Parent CLASS properties were not properly initialized, you might end up with unexpected behavior in your Derived CLASS. Therefore, Automatic Constructors are not automatically overridden. Instead, by default, they all execute in order when the object is instantiated.

First, the Parent CLASS Constructor executes, then the Derived CLASS Constructor executes. They execute in the order of their derivation. Base CLASS Constructors always execute first, followed by any Derived CLASS Constructors, in the order in which they were derived. The Constructor for the most derived CLASS always executes last.

The same reasoning is true for Automatic Destructors, except the order of their execution is reversed. The most derived CLASS Destructor automatically executes first when the object is destroyed, and on down the chain of derivation until the Base CLASS Destructor executes last. In other words, LIFO: Last In First Out

This is all pretty standard. The way Automatic Constructors and Destructors work by default in the Clarion language is the same way they work in most other OOP languages. However, Clarion does give you some flexibility that some other OOP languages do not.

I told you that Automatic Constructors and Destructors are not automatically overridden in Clarion. The key word here is "automatically." You CAN override them in Clarion if you want to – something you can't do in some other OOP languages. If you add the REPLACE attribute to the prototype of your CONSTRUCT or DESTRUCT method, you are telling the compiler that you DO want to override the method.

So, what does that buy you? Suppose that for some reason, you need to initialize a variable in your Derived CLASS before the Parent CLASS Constructor executes? The only way to do that is to override the Constructor. You simply place the REPLACE attribute on the Derived CLASS CONSTRUCT method's prototype then write your code.

And now you'll ask, "What about your great argument against automatically overriding Constructors and Destructors? Have you forgotten that?"

The argument against *automatically* overriding Constructors and Destructors is still valid, and needs to be considered carefully when you decide to *explicitly* override them. But we have thought of that.

PARENT

You recall that the way to reference the current object within a method's code is to use SELF instead of the object name? Well, we've provided another tool in Clarion's OOP syntax that allows you to call a method from a Parent CLASS even when you've overridden that method.

Prepending PARENT to the method name explicitly calls the Parent CLASS's method, even though it has been overridden. This technique holds true not only for Constructors and Destructors, but also for *any* overridden methods.

```
 PROGRAM

MyClass   CLASS,TYPE     !Declare Base Class
Property    LONG
Method      PROCEDURE
Construct   PROCEDURE
          END

ClassA    CLASS(MyClass) !Declare Derived Class
Aproperty  LONG
Construct  PROCEDURE,REPLACE
          END
 CODE                    !ClassA Instantiation here

ClassA.Construct PROCEDURE
 CODE
 SELF.Aproperty = 1      !Initialize then call
 PARENT.Construct        ! parent constructor
```

Therefore, when you need to explicitly control the execution order of your Constructors or Destructors, you simply put the REPLACE attribute on the prototype, then in your Constructor or Destructor method's code directly call PARENT.CONSTRUCT or PARENT.DESTRUCT at the exact point within your logic that is most appropriate to what you need to do. That could be before, after, or in the middle of your Derived CLASS's code – wherever you need it to be.

Polymorphism

OK, there's one last, really major OOP buzzword you have to learn: Polymorphism. This is a really big one, and it's the key that makes Object Oriented Programming so powerful. So let's start with the Oxford English Dictionary definition of Polymorphism: *The condition or character of being polymorphous; the occurrence of something in several different forms.*

Now the Clarion language has had some forms of polymorphism for a long time already. For example, the *? and ? parameter types which indicate an unknown data type for parameter passing allow you to write polymorphic procedures. Clarion has had these since the DOS days.

In Clarion for Windows version 2.0 we introduced Procedure Overloading which allows you to create multiple procedures using the same name but taking different parameter lists – another form of polymorphism.

OK, right about now all the OOP purists in the audience are saying to themselves, "That's got nothing to do with Polymorphism in OOP," and they're right.

All that polymorphic functionality that we've given you in previous versions of Clarion, going all the way back to the original DOS 1.0 version, has nothing at all to do with what OOP means when it talks about Polymorphism.

Virtual Methods

There's one more common OOP term that is virtually synonymous with Polymorphism: Virtual Methods.

Now this is the point where OOP starts to get a little tricky, so you guys that are new to OOP – don't worry if you don't "get it" right away from this morning's talk. This is the part that causes people to say that it takes at least six months to really "get" OOP and how to program in it.

Whether you already understand OOP or not, the ABC Library is an excellent example of well-designed classes that you can study and learn from.

Virtual Methods are what OOP purists are really talking about when they say Polymorphism. What's a Virtual Method?

The simple answer to that is: A Virtual Method is a method whose prototype is present in both a Parent and Derived CLASS structure, and has the VIRTUAL attribute on both.

This means that Virtual Methods must have the exact same prototype in both a Parent and Derived CLASS, and that they both must have the VIRTUAL attribute. That's the mechanics of Virtual Methods.

So, now that you've gotten the mechanics of Virtual Methods down, we can go on to the really important stuff, like – what good are they and what do they do?

You recall that a Derived CLASS object inherits the methods of its Parent CLASS? That effectively means that a Derived CLASS can "call down" to execute a Parent CLASS method.

Well, Virtual Methods are the opposite – they allow Parent CLASS methods to "call up" to execute a Derived CLASS method.

To repeat, Virtual Methods allow the code in Parent CLASS methods to execute Derived CLASS methods.

More Apple Pie

The easiest way to see how this works is with an example. Let's go back to our ApplePie classes. We'll add a PreparePie method to the Base CLASS because every type of Apple Pie needs preparation. Some other common tasks that the code in PreparePie will need to execute will be CreateCrust and MakeFilling, so we'll prototype methods for each of these tasks.

```
ApplePie   CLASS,TYPE
PreparePie     PROCEDURE
CreateCrust    PROCEDURE,VIRTUAL   !Virtual Methods
MakeFilling    PROCEDURE,VIRTUAL
           END

Dutch     CLASS(ApplePie)
CreateCrust    PROCEDURE,VIRTUAL    !Virtual Methods
MakeFilling    PROCEDURE,VIRTUAL
           END
 CODE
 Dutch.PreparePie        !Will call the Dutch object's Virtuals

ApplePie.PreparePie PROCEDURE
 CODE
 SELF.CreateCrust
 SELF.MakeFilling
```

These two methods are VIRTUAL because the actual code to accomplish them will probably be a little different for each type of pie we derive. However, we do know that these are always the steps you need to take to prepare any apple pie, so we can write that code into the Base CLASS PreparePie method and know that it won't change.

Now we'll derive the *Dutch* ApplePie CLASS. Notice that the prototypes for CreateCrust and MakeFilling are exactly the same as in the Base CLASS, including the VIRTUAL attribute.

Now, you execute the Dutch.PreparePie method. This is an inherited method, so the Derived CLASS is "calling down" to its Parent CLASS method – ApplePie.PreparePie. That means the code that executes is the code inside the ApplePie.PreparePie method.

The first statement that executes in the PreparePie method is the call to SELF.CreateCrust. So, the question is, which CreateCrust method will actually execute?

At this point, we have two defined: the one in the *ApplePie* Base CLASS, and the other in the *Dutch* Derived CLASS. The answer is, the actual method that will execute is the Dutch.CreateCrust method, because the original call to PreparePie referenced the *Dutch* object. The ApplePie Base CLASS PreparePie method is "calling up" the chain of derivation to the Derived CLASS's CreateCrust method.

OK, so now you see how it all works — Virtual Methods are sort of "reverse inheritance."

The Base CLASS code calls a method and, at the time the code is compiled, it doesn't have any idea at all what actual code will need to execute as a result of the Virtual Method call. Because of the prototype in the Base CLASS, it knows there is a method of that name that it can call to perform a specified task, it just doesn't know exactly how that code will do its task – and it doesn't care. The ApplePie Base CLASS tells the current object to create its crust and it goes away and does it.

Now, I know some you are asking yourselves, "How can this kind of code compile? I remember my compiler class in college and the compiler needs to resolve all procedure calls at compile time." The answer is using another common OOP technique: Late Binding.

Late Binding

Early binding is the kind of stuff you were taught in your compiler theory class where all procedure calls are resolved at compile time. However, since calls to Virtual Method cannot be resolved at compile time, the compiler must build what's called a "Virtual Method Table" at compile time and set the method call up for Late Binding. At run time, when the call to the Virtual Method executes, Late Binding means that the actual method to call is determined by a lookup into the Virtual Method Table. I'm sure that all of you know a lot about lookups, since it's a pretty standard database application design technique!

Now, in many other OOP languages this Late Binding for Virtual Methods can cause a real performance hit. However, in Clarion the entire Late Binding process takes only one more extra reference at runtime than Early Binding does, so there's *virtually* no performance hit with using Virtual Methods in Clarion.

As a matter of fact, Virtual Methods in Clarion are so efficient that, when you look at the code generated by the new ABC Templates you'll find that almost all the code generated is now in Virtual Methods. This means you've probably seen the last of any Pool Limit errors!

That brings me to the last OOP issue I want to tell you about this morning — scoping. In older version of Clarion, you will have noticed that some of the biggest conversion problems lay in the fact that many of the template embed points were moved into Virtual Methods. This meant you had two workarounds available to you: move any procedure local data that you wanted to reference in those embed points out of the PROCEDURE itself and into Module data; or jump through some even more difficult hoops by creating reference variables for each of the variables you wanted to reference and reference assigning to them, then changing your embed code to refer to those reference variables. Needless to say, this was a lot more work than Clarion Programmers have grown accustomed to!

So, to solve that problem, Clarion supports a technique called Local Derived Methods.

Local Derived Methods

Local Derived Methods are declared in a Derived CLASS structure within a PROCEDURE. The Local Derived Method definitions (their executable code, that is) must immediately follow the end of the PROCEDURE in which they are declared.

So, what's the advantage you get from these things? Local Derived Methods inherit the scope of the PROCEDURE within which they are declared. That means that all the PROCEDURE's Local Data variables and ROUTINEs are visible and available inside the Local Derived Methods. That eliminates all the hoops you had to jump through in older versions to access your local data in those virtual embed points. And it also enables you to call the PROCEDURE's ROUTINEs from within them, too, just as if your code were still within the PROCEDURE itself.

By the way, for those of you who still remember some Pascal from college, this concept is very similar to Pascal's nested procedures.

This new implementation of scoping for Local Derived Methods is what allowed great flexibility and functionality in the ABC Templates and ABC Library. All you need to do is run the App Wizard on any dictionary then look at the generated code to see that most generated PROCEDUREs now contain very little code – everything has been moved into Local Derived Methods. And you will mostly still be able to write your embed code as if it were inside the PROCEDURE itself.

Summary

You've now been exposed to the three major OOP buzzwords: Encapsulation, Inheritance, and Polymorphism; and how we've implemented them in the Clarion language.

You've also heard most of the other standard OOP terms: Properties, Methods, Objects, Instantiation, Base Classes, Derived Classes, SELF, PARENT, Constructors and Destructors, Virtual Methods, Late Binding. . .

And let's not forget the one you do NOT have to learn: Disambiguate!

3 - Object Oriented Programming (OOP)

Object Overview - What are Objects?

An object is a single “thing.”

We humans tend to group what we see in the world into classes of things. We group them according to the properties and behaviors that they have in common. Therefore, things that belong to a single class have properties and behaviors that describe the type of thing it is and what it does.

We also tend to divide the classes of things we see into a tree-shaped hierarchy of classes. The hierarchy tree starts with the most general at the bottom and grows up to the most specific. Each branch of the tree is a separate class that shares the common properties of the class above it to which it is attached (its parent) because it is derived from the parent. Each derived class must also have characteristics that are unique to it that separate it from the parent and from other classes derived from the same parent.

An object is one instance of a class of things, generally from the lowest level of the hierarchy tree. It has properties and behaviors that define what it is. For example, we see two major classes of things in our world: Living Things, and Non-living Things. We can divide the Living Things class into two sub-classes: Plants and Animals. Within the Animal class we can see many sub-classes: Mammals, Birds, Fish, etc. Then these sub-classes are further sub-divided into sub-sub-classes... This sub-division of classes goes on until you get all the way to the individual object (a single “thing”).

Why Objects?

Object-oriented programming (OOP) techniques were developed to let us more closely model the programs we write on the way we look at the real world. Designing a class hierarchy to solve a real-world problem is a matter of looking at the problem from the same perspective that we look at the natural world. We start at the most abstract—general classes that have the properties and behaviors common to all members of the class—then derive from those abstract classes the specific classes that fully describe the set of individual objects in the problem.

One major benefit of object-oriented programming is extensive code re-use. Once a behavior (method) has been coded for the more general class, it never needs to be re-written for the derived classes in which it does not need to change. Therefore, it is written once and the same code is shared amongst all the objects derived from that class.

What Makes an Object?

There are three major concepts in object-orientation: encapsulation, inheritance, and polymorphism. The Clarion language contains object-oriented extensions that cover all these concepts.

Encapsulation

Encapsulation means bundling the properties of a class (its data members) together with its methods (the procedures that operate on the data members) into one coherent unit. The most important benefit of encapsulation is the ability to treat the object as a single complete entity. This lets us use the object without knowing everything about it. It also lets us change one object without affecting other unrelated objects.

The properties of an object are the things the object knows about itself, and the methods are its behaviors—the operations it can perform. Each object has its own properties unique to itself, and it shares the same methods with all the other objects of the same type (other object instances of the same class).

For example, objects that belong to a class (such as the class “normal healthy human beings”) each have their own unique set of properties (like blue eyes or brown eyes) but they all share the same basic abilities common to the class of objects to which they belong (vision).

Inheritance

Inheritance is the mechanism that allows us to build hierarchies of classes. A derived class inherits all the properties and methods of the class from which it is derived (its base class). This provides the derived class with a “starting point” of all the common properties and methods of the more general (abstract) class. Then the derived class can add the properties and methods that differentiate it from its parent.

Inheritance is also a core element of object-orientation that provides for code re-use. The code for inherited methods that are not overridden in the derived class exists only once—in the class in which they are defined.

Polymorphism

Polymorphism generically means being able to call a method that operates differently depending on how it is called. For example, the Clarion OPEN statement is polymorphic because it performs different operations depending on whether it is given a FILE or a WINDOW to open.

This type of polymorphism is generally called “Procedure overloading” because you are “overloading” what appears to be one procedure call with multiple operations. Actually, procedure overloading is simply done in Clarion by defining separate procedures with the same name but different parameter lists. See Procedure Overloading in the Language Reference for a complete discussion of this topic.

Polymorphism in object-oriented parlance is more commonly taken to mean the ability of a base class method to call methods of classes derived from the base class—without knowing at compile-time exactly what method is actually going to be called. This is called using “Virtual Methods.” If you look at inheritance as the derived class being able to “call down” to base class methods, then you can also look at virtual methods as the base class being able to “call up” to derived class methods. This seems like a bit of magic, since the base class can never know what classes have been derived from it.

To handle virtual methods, the compiler must implement “late binding” instead of “early binding” when creating the executable. With early binding, the compiler can resolve a non-virtual procedure call at compile time to a specific code address in the executable. This means a direct call to the procedure, which is very efficient.

However, with late binding, the compiler must, at compile time, create a Virtual Method Table (VMT) that contains the specific code addresses of all the virtual methods. It then must insert code that resolves, at run time, each call to a virtual method by first looking for the method in the VMT, then calling the appropriate method. This may seem like it would create a fairly large performance hit, but with the highly efficient Clarion compiler, it is actually almost as fast as calling a non-virtual method.

This virtual method type of polymorphism is the mechanism that allows us to handle the individual differences between classes derived from the same base class while also allowing the base class to ignore those differences. This lets a common method in a base class call a method from one of its derived classes to specifically handle some operation in a manner appropriate to the specific derived class.

For example, suppose we have a Vehicle class, which contains a virtual method to Steer the vehicle, because all vehicles must have some way to steer themselves. When you derive a Bicycle class from the Vehicle class, the derived class contains its own specific Steer virtual method appropriate to a Bicycle. Then when you derive a Car class from the Vehicle class, it also contains its own specific Steer virtual method for a Car. Of course, both the Bicycle and Car derived classes also inherit a common Move method which calls the Steer method. The inherited Move method does not know or care whether it’s actually calling the Car’s Steer method or the Bicycle’s Steer method at runtime, it just tells the object (whether it’s a Car or a Bicycle) to Steer itself (and it does).

Clarion's OOP Extensions

The specific Clarion language syntax that allows you to declare classes of objects and derive classes from previously declared base classes starts with the CLASS structure.

The CLASS Structure—Encapsulation

The Clarion CLASS structure declares an object class containing properties and declaring the methods that operate on those properties. In Clarion, the properties of the class are the data members declared in the CLASS and methods are the PROCEDUREs prototyped in the CLASS structure.

This example declares a very simple CLASS structure:

```
SomeClass         CLASS,MODULE('SomeClas.CLW')
PropertyA          LONG           !Data member (property)
PropertyB          LONG
ManipulateAandB    PROCEDURE      !Method
                  END
```

This CLASS is named SomeClass and it contains two properties (data members): PropertyA and PropertyB. It also contains one method: the ManipulateAandB PROCEDURE. The MODULE attribute on the CLASS statement specifies that the code defining the ManipulateAandB PROCEDURE exists in the SomeClas.CLW file.

CLASS Properties

The properties (data members) that a CLASS may contain are limited to the data types that are appropriate to have in a GROUP structure. This means that all the simple data types are allowed (LONG, SHORT, REAL, etc.) including GROUP structures, but complex data types (FILE, QUEUE, WINDOW, etc.) are not allowed. Allowing only simple data types in a CLASS may seem like a limitation, but it actually is not because Reference variables are also allowed (more on this later).

CLASS Methods

The methods (PROCEDUREs) declared in a CLASS are defined in the source file named in the MODULE attribute. This encapsulates all the methods in a single source file, making maintenance much easier, and also offers some other advantages that we will get to later.

For the CLASS declaration above, the SomeClas.CLW file would contain code similar to the following:

```
SomeClass.ManipulateAandB  PROCEDURE
 CODE
 MESSAGE('A = ' & SELF.PropertyA)
 MESSAGE('B = ' & SELF.PropertyB)
```

The label of the PROCEDURE statement begins with the name of the CLASS to which the method belongs, prepended to the name of the method, and connected with a period. You can also label the method and name the CLASS to which it belongs as an implicit first parameter (and labeling the classname SELF), like this:

```
ManipulateAandB PROCEDURE(SomeClass SELF)
 CODE
 MESSAGE('A = ' & SELF.PropertyA)
 MESSAGE('B = ' & SELF.PropertyB)
```

Within a method, you use the SELF keyword, not the CLASS label, to address a property or method of the current instance of the CLASS (the current object) on which the method is operating.

Creating Objects

An object is one specific instance of a CLASS containing its own set of properties, specific to that one instance. All object instances of a CLASS share all the methods of the CLASS, so the methods exist only once each.

A CLASS declaration with the TYPE attribute only declares the CLASS—there are no objects until you create an instance of the CLASS (also called instantiation). A CLASS declaration without the TYPE attribute both declares the CLASS and instantiates it (creates the first object).

There are two ways to create an object: declare it in a data section, or create one dynamically in executable code using NEW. To declare an object in the data section, simply name the CLASS as the data type:

```
  PROGRAM
SomeClass          CLASS               !Creates a CLASS and an instance
PropertyA            LONG
PropertyB            LONG
ManipulateAandB      PROCEDURE
                   END
MyClass            CLASS,TYPE        !Creates just a CLASS
PropertyA            LONG
PropertyB            LONG
ManipulateAandB      PROCEDURE
                   END
SomeClassObject1   SomeClass         !Create an instance of SomeClass
SomeClassObject2   SomeClass         !Another instance of SomeClass
MyClassObject1     MyClass           !Create an instance of MyClass
MyClassObject2     MyClass           !Another instance of MyClass
  CODE
```

An object declared in a data section is automatically destroyed for you when it goes out of scope. For example, an object declared in a PROCEDURE's local data section goes out of scope (and is destroyed) upon RETURN from the PROCEDURE. Objects declared in a global or module data section go out of scope (and are destroyed) only when the program terminates.

To create an object dynamically in executable code using NEW, declare a reference variable for the CLASS, then execute a reference assignment statement to the reference variable, naming the CLASS as the parameter to NEW:

```
  PROGRAM
SomeClass            CLASS,TYPE
PropertyA              LONG
PropertyB              LONG
ManipulateAandB        PROCEDURE
                     END
SomeClassObjectRef  &SomeClass          !A Reference to a SomeClass object
  CODE
  SomeClassObjectRef &= NEW(SomeClass) !Create the object
```

It is important to note that when you use NEW to instantiate the object, you must also use DISPOSE to destroy it when it goes out of scope—the object is not automatically destroyed for you. The advantage of using NEW is the ability to place the reference variable that "points at" the object inside another object as a property.

Using References as Properties

Since reference variables are valid to use anywhere the label of the data type they reference may be used, they allow a CLASS to access all the types of complex structures that cannot be directly declared inside the CLASS. When you couple with this the use of NEW and DISPOSE to dynamically create and destroy variables and objects, you give a CLASS the ability to contain almost all these complex structures.

For example, using a reference variable in the CLASS, you can declare a reference to a queue (&QUEUE) that is passed to the CLASS methods. The same thing is true of all the complex data types (&QUEUE, &FILE, &KEY, &BLOB, &VIEW, or &WINDOW). Although the CLASS cannot directly contain the data declaration of these complex structures, the CLASS methods can act as if they "own" the structure just as surely as the instance "owns" its own set of properties (data members). For example, although a FILE declaration must be external to a CLASS structure, by declaring a reference variable for the file (&FILE) within the CLASS, then executing a reference assignment statement for a specific FILE structure to an object's (one specific instance of the CLASS) reference variable, the object can "own" the FILE and its methods can execute any statement that requires the label of a FILE structure as a parameter, effectively operating directly on the FILE structure itself.

You can also declare a reference to a specific type of QUEUE, GROUP, or CLASS structure (&QueueName, &GroupName, or &ClassName) which a CLASS method can then dynamically create. This allows the CLASS to contain specific types of QUEUEs and other named CLASSes without the need to declare them within the CLASS structure.

This example declares a simple CLASS structure containing a reference variable that “points to” a specific type of QUEUE structure:

```
MyQueue         QUEUE,TYPE          !Define a specific type of queue
Field1           LONG
Field2           STRING(20)
                END
SomeClass       CLASS
QueRef           &MyQueue           !A queue with a LONG and a STRING(20)
CreateQue        PROCEDURE          !Create the queue for each object
                END
SomeClass.CreateQue  PROCEDURE
  CODE
  SELF.QueRef &= NEW(MyQueue)       !Create a QUEUE for the current object
  SELF.QueRef.Field1 = 1
  SELF.QueRef.Field2 = 'First Entry'
  ADD(SELF.QueRef)                  ! and add the first entry to the queue
  IF ERRORCODE() THEN STOP(ERROR()).
```

Using these “named” references and naming a reference to a specific CLASS structure makes it possible for one object to effectively “contain” an instance of another object. This is an OOP technique known as “composition” which provides an alternative to multiple inheritance (which we will get to shortly when we discuss inheritance).

Constructors and Destructors

Constructors and destructors are a feature of many object-oriented languages. These are methods (which you must write) that execute automatically, without being explicitly called. A constructor executes when the object is created, and a destructor executes when the object is destroyed.

The most common purpose of a constructor method in other object-oriented languages is to allocate memory for the object and initialize data members to prevent bugs that could be caused by uninitialized variables. Destructor methods in other object-oriented languages usually just de-allocate the memory for the object. The Clarion language automatically allocates and de-allocates memory and initializes variables to blank, zero, or the value named in the variable’s declaration. This means the biggest reason for supporting automatic constructors and destructors doesn’t exist in Clarion. However, Clarion does support automatic constructors and destructors anyway.

If a CLASS contains a method named Construct (which must be a PROCEDURE that takes no parameters), then that Construct method is automatically called when the object is instantiated. If a CLASS contains a method named Destruct (also a PROCEDURE that takes no parameters), then that Destruct method is automatically called when the object is destroyed. Since Clarion automatically takes care of memory allocation and initialization, the most common use of constructors and destructors in a Clarion program would be to initialize and dispose of an object's reference variables.

```
SomeClass      CLASS,TYPE
ObjectQ          &MyQueue       !Reference to a specific type of QUEUE
Construct        PROCEDURE      !Constructor method
Destruct         PROCEDURE      !Destructor method
                 END
ClassRef      &SomeClass        !Reference to a SomeClass object
  CODE
  ClassRef &= NEW(SomeClass)    !Create a SomeClass object
                                ! which auto-calls its constructor method
  DISPOSE(ClassRef )              !Destroy the SomeClass object
                                ! which auto-calls its destructor method

SomeClass.Construct  PROCEDURE
  CODE
  SELF.ObjectQ &= NEW(MyQueue)  !Create the QUEUE

SomeClass.DEstruct  PROCEDURE
  CODE
  FREE(SELF.ObjectQ)            !Free any QUEUE entires
  DISPOSE(SELF.ObjectQ)         !Destroy the QUEUE
```

In Clarion (unlike many other object-oriented languages), you may also explicitly call the constructor and destructor methods. This allows you to "re-start" an object at any time by simply calling the object's destructor (clearing it) then its constructor (re-initialize it).

Public vs. PRIVATE vs. PROTECTED

All the properties and methods of a CLASS are public unless explicitly declared with the PRIVATE or PROTECTED attribute. In this case, "public" means they are visible and available for use anywhere that the object is in scope.

PRIVATE

The PRIVATE attribute specifies that the property or method on which it is placed is visible only to the PROCEDUREs defined within the source module containing the methods of the CLASS structure. This completely encapsulates the data and methods from other CLASSes.

```
SomeClass    CLASS,MODULE('SomeClas.CLW'),TYPE
PublicVar     LONG                          !Declare a Public property
PrivateVar    LONG,PRIVATE                  !Declare a Private property
BaseProc      PROCEDURE(REAL Parm)          !Declare a Public method
Proc          PROCEDURE(REAL Parm),PRIVATE  !Declare a Private method
             END

TwoClass    SomeClass                       !Declare an object
  CODE
  TwoClass.PublicVar = 1                    !Legal assignment
  TwoClass.PrivateVar = 1                   !Illegal assignment
  TwoClass.Proc(2)                          !Illegal call to Proc
```

!The SomeClas.CLW source code file contains:

```
  MEMBER('MyProg')
  MAP                                   !A local MAP which declares
SomeLocalProc    PROCEDURE(SomeClass)   !a "friend" of SomeClass and
                                        !passing current object to it
  END

SomeClass.BaseProc    PROCEDURE(REAL Parm)
  CODE
  SELF.PrivateVar = Parm                !Legal assignment
  SELF.Proc(Parm)                       !Legal call to Proc
  SomeLocalProc(SELF)
               !Call the friend, passing it the current object instance

SomeClass.Proc        PROCEDURE(REAL Parm)
  CODE
  RETURN(Parm)

SomeLocalProc    PROCEDURE(PassedObject)   !Visible only in this module
  CODE
  PassedObject.PrivateVar = 1              !Legal assignment
```

A side benefit of the Clarion implementation of PRIVATE properties and methods are friends. In C++, the concept of a friend is a procedure that, while not a method of a class, shares the private properties of the class. In the above example, SomeLocalProc is a friend of OneClass because it is defined in the same source code module as the methods that belong to OneClass, which gives it access to the PRIVATE data members. Passing the current object to SomeLocalProc enables it to directly address the private properties of the current object (the current instance of the CLASS).

PROTECTED

The PROTECTED attribute specifies that the property or method on which it is placed is visible only to the PROCEDUREs defined within the source module containing the methods of the CLASS structure and the methods of any CLASS derived from that CLASS. This only encapsulates the data or method from other CLASSes not derived from the CLASS in which it was declared (or any other non-CLASS code).

The purpose of the PROTECTED attribute is to provide a level of encapsulation between Public and PRIVATE. PROTECTED data or methods are available for use within their own CLASS and derived CLASSes, but not available to any code outside those specific CLASSes.

Derived CLASSes—Inheritance

Clarion supports inheritance—one CLASS declaration can be built on the foundation of another CLASS's properties and methods. The CLASS from which a CLASS is derived is generally referred to as its base class. The derived CLASS structure inherits all the public and PROTECTED data members and methods of the base class as a starting point (but not the PRIVATE data members and methods). This means the inherited data members and methods are visible and available for use within the methods declared within the derived CLASS structure.

This example declares one CLASS structure derived from another:

```
SomeClass         CLASS               !Declare a base class
PropertyA           LONG
PropertyB           LONG
ManipulateAandB     PROCEDURE
                  END

AnotherClass     CLASS(SomeClass)     !Derived from SomeClass
PropertyC          LONG
ManipulateAndC     PROCEDURE
                 END
```

In this simple example, AnotherClass is derived from SomeClass (its base class), inheriting all the properties and methods that exist in SomeClass. AnotherClass also declares one new property and one new method that don't exist in SomeClass. Therefore, objects that belong to SomeClass have two properties and one method while objects that belong to AnotherClass have three properties and two methods.

Given the two CLASS declarations above, the following executable statements are all valid:

```
CODE
SomeClass.PropertyA = 10            !Assign values to the properties
SomeClass.PropertyB = 10
SomeClass.ManipulateAandB           !Call the object's method

AnotherClass.PropertyA = 10         !Assign values to the properties
AnotherClass.PropertyB = 10
AnotherClass.PropertyC = 10
AnotherClass.ManipulateAandB       !Call the object's methods
AnotherClass.ManipulateAndC
```

Also given the declarations above, the following executable statements are not valid:

```
SomeClass.PropertyC = 10  !Invalid, object does not have this property
SomeClass.ManipulateAndC  !Invalid, object does not have this method
```

These statements are invalid because, although the derived object contains all the properties and methods of the CLASS from which it is derived, the reverse is not true—inheritance is one-way. There is a mechanism that gets around this and allows base class methods to call derived class methods (virtual methods) that we will get to later.

Overriding Inherited Methods

There are circumstances where a derived class may need to override an inherited method to provide the explicit functionality required for that specific class of object. This is simple to accomplish in Clarion, just re-declare the method in the derived class (with exactly the same parameter list) then re-define the method in the derived class for the new functionality.

This example overrides an inherited method:

```
SomeClass        CLASS
PropertyA          LONG
PropertyB          LONG
ManipulateAandB    PROCEDURE
                   END

AnotherClass    CLASS(SomeClass)
PropertyC          LONG
ManipulateAandB    PROCEDURE               !Override the inherited method
ManipulateAndC     PROCEDURE
                END

SomeClass.ManipulateAandB  PROCEDURE
Product    LONG
  CODE
  Product = SELF.PropertyA * SELF.PropertyB
  MESSAGE('Product of A*B is ' & Product)

AnotherClass.ManipulateAandB  PROCEDURE    !Override the inherited method
DoubleProduct    LONG
  CODE
  DoubleProduct = (SELF.PropertyA * SELF.PropertyB) * 2
  MESSAGE('Double the Product of A*B is ' & DoubleProduct)

AnotherClass.ManipulateAndC  PROCEDURE
DoubleProduct    LONG
  CODE
  DoubleProduct = (SELF.PropertyA * SELF.PropertyC) * 2
  MESSAGE('Double the Product of A*C is ' & DoubleProduct)
```

In this example, AnotherClass overrides the ManipulateAandB method with its own version containing a slightly different algorithm. Notice that the parameter lists are the same (neither method receives parameters). This is the key to overriding methods in derived classes; the parameter lists must be the same.

If the parameter lists of the two ManipulateAandB methods in this example were different, then you would have procedure overloading in AnotherClass; that is, AnotherClass would actually contain two methods named ManipulateAandB and the compiler would have to resolve which one to call by means of the differing parameters. See Function Overloading in the Language Reference for a full discussion of this technique.

Simply overriding a method in a derived class does not automatically mean you cannot also call the base class method from within derived class methods. Prepending PARENT to the method call (like SELF) allows a derived class to explicitly call base class methods (PARENT.MethodName). This allows you to "incrementally" override methods without requiring that you duplicate the base class code in the derived class method that overrides it (assuming you still need the base class method's functionality).

This example overrides an inherited method and calls the base class method from the derived class:

```
SomeClass         CLASS
PropertyA           LONG
PropertyB           LONG
ManipulateAandB     PROCEDURE
                  END

AnotherClass     CLASS(SomeClass)
ManipulateAandB     PROCEDURE
                 END

SomeClass.ManipulateAandB  PROCEDURE
Product    LONG
  CODE
  Product = SELF.PropertyA * SELF.PropertyB
  MESSAGE('Product of A*B is ' & Product)

AnotherClass.ManipulateAandB  PROCEDURE
DoubleProduct    LONG
  CODE
  PARENT.ManipulateAandB                    !Call the PARENT method first
  DoubleProduct = (SELF.PropertyA * SELF.PropertyB) * 2
  MESSAGE('Double the Product of A*B is ' & DoubleProduct)
```

For constructor and destructor methods, simply re-declaring the Construct or Destruct method in the derived class does not override the inherited method unless the REPLACE attribute is specified on the method prototype. Without the REPLACE attribute, a derived class constructor automatically calls the base class constructor first (before the derived class constructor executes), and a derived class destructor automatically calls the base class destructor last (after the derived class destructor executes).

If the REPLACE attribute is present on the Construct or Destruct method's prototype, then the derived class method does not automatically call the base class method. However, the base class method can be called from within the derived class method's code by explicitly calling PARENT.Construct or PARENT.Destruct. This allows you to "incrementally" customize derived class constructors or destructors without completely re-writing the base class code.

```
MyQueue        QUEUE,TYPE          !Declare a type of QUEUE with one field
Field1          STRING(10)         ! containing 10 bytes of string data
               END

SomeClass      CLASS,TYPE          !Base class
ObjectQ          &MyQueue
Construct        PROCEDURE         !Constructor method
Destruct        PROCEDURE          !Destructor method
               END
AnotherClass  CLASS(SomeClass)     !Derived class
Construct        PROCEDURE,REPLACE!Override the Constructor
               END
ClassRef      &AnotherClass        !Reference to a SomeClass object
  CODE
  ClassRef &= NEW(AnotherClass)    !Create a AnotherClass object which
                                   ! calls the derived class constructor
  DISPOSE(ClassRef)                !Destroy the object
                                   ! which calls the base class destructor

SomeClass.Construct  PROCEDURE     !Base class constructor
  CODE
  SELF.ObjectQ &= NEW(MyQueue)

SomeClass.Destruct  PROCEDURE      !Base class destructor
  CODE
  FREE(SELF.ObjectQ)
  DISPOSE(SELF.ObjectQ)

AnotherClass.Construct  PROCEDURE !Derived class constructor
MyString    STRING(10),DIM(10)
  CODE
  LOOP X# = 1 TO 10                !Do some preliminary work, then
    MyString[X#] = 'Entry ' & FORMAT(X#,@N02)
  END

  PARENT.Construct                 ! call the base class constructor
  LOOP X# = 1 TO 10                ! then finish off the construction
    SELF.ObjectQ.Field1 = MyString[X#]
    ADD(SELF.ObjectQ)
    ASSERT(~ERRORCODE())
  END
```

Multiple Inheritance vs. Composition

Single inheritance means that a derived class has only one base class from which it inherits properties and methods. Clarion directly supports single inheritance. Some OOP languages (most notably, C++) allow multiple inheritance, wherein a derived class inherits properties and methods from several classes. This has the advantage of easily combining existing classes to create derived classes. It also has the drawback that the compiler must deal with a lot of potential ambiguity if two or more of the classes from which the new class is derived contain methods with the same name—you must write extra code to disambiguate the overloaded methods.

Although Clarion only supports single inheritance (like many other OOP languages) you can easily get around this limitation using a standard OOP technique called “Composition.” Composition means to place an object of one class within another. Composition provides the benefits of multiple inheritance without the potential ambiguity. For those instances where you need multiple inheritance, simply decide which class to derive from and place an object of the other class (the contained object) in the new class (the container object). You then need to implement a constructor method to instantiate the contained object.

You implement composition in Clarion by placing a Reference to the object in the CLASS declaration, like this:

```
  PROGRAM
SomeClass         CLASS,TYPE
PropertyA           LONG
PropertyB           LONG
ManipulateAandB     PROCEDURE
                   END

AnotherClass     CLASS,TYPE
PropertyC           LONG
ManipulateAndC    PROCEDURE
                 END

MultiClass       CLASS(SomeClass),TYPE  !Inherits from SomeClass and
AnotherClassRef     &AnotherClass        ! contains an AnotherClass object
Construct           PROCEDURE            ! and a Constructor method
                 END

MClass           MultiClass            !Declare an object
MClassRef        &MultiClass           !Declare an object reference

  CODE               !The constructor Instantiates the contained object
                     ! when the container object comes into scope

  MClassRef &= NEW(MultiClass)         !Create a new container object
                                       ! which auto-calls its constructor

MultiClass.Construct  PROCEDURE
  CODE
  SELF.AnotherClassRef &= NEW(AnotherClass)  !Create the contained object
```

This same technique also gives you recursive classes by placing a Reference to an object of the same CLASS in the declaration, like this:

```
SomeClass        CLASS,TYPE          !SomeClass recurses into itself by
SomeClassRef      &SomeClass         ! containing a SomeClass object
PropertyA           LONG
PropertyB           LONG
ManipulateAandB     PROCEDURE
Construct           PROCEDURE
                 END
```

Virtual Methods—Polymorphism

Inheritance allows a derived class to “call down” to the methods it has inherited from its base class. Virtual methods, on the other hand, allow the methods in a base class to “call up” to methods in derived classes, despite not knowing exactly what is being called. To accomplish this, you must prototype the virtual method in both the base class and the derived class.

The VIRTUAL attribute on a method’s prototype declares a virtual method. This attribute must appear on the method’s prototype in both the base class and the derived class. Usually, the base class definition of the virtual method is a dummy procedure (one that does nothing) or one that provides some simple default functionality for those derived classes that don’t need their own method definition.

This example defines two simple virtual methods in two derived classes:

```
SomeClass         CLASS,TYPE
PropertyA           LONG
PropertyB           LONG
InitAandB           PROCEDURE(LONG PassedA, LONG PassedB)
ManipulateAandB     PROCEDURE,VIRTUAL       !Declare base method virtual
                  END
AnotherClass     CLASS(SomeClass),TYPE
ManipulateAandB     PROCEDURE,VIRTUAL       !Re-declare the virtual method
                 END
DifferentClass   CLASS(SomeClass),TYPE
ManipulateAandB     PROCEDURE,VIRTUAL       !Re-declare the virtual method
                 END

Object1           AnotherClass              !Declare an object
Object2           DifferentClass            !Declare a diferent object

  CODE
  Object1.InitAandB(10,20)  !InitAandB will call the AnotherClass method
  Object2.InitAandB(30,40)  !InitAandB will call the DifferentClass
method

SomeClass.InitAandB  PROCEDURE(LONG PassedA, LONG PassedB)
  CODE
  SELF.PropertyA = PassedA
  SELF.PropertyB = PassedB
  SELF.ManipulateAandB                !Call whichever virtual method is
                                      ! appropriate for the current
object

SomeClass.ManipulateAandB  PROCEDURE  !Base class method does
  CODE                                ! nothing

AnotherClass.ManipulateAandB  PROCEDURE            !Virtual method
  CODE
  MESSAGE(‘The Product of A*B is ‘ & (SELF.PropertyA * SELF.PropertyB))
DifferentClass.ManipulateAandB  PROCEDURE          !Virtual method
  CODE
  MESSAGE(‘The Sum of A+B is ‘ & (SELF.PropertyA + SELF.PropertyB))
```

In this example, the ManipulateAandB method is *virtual*. The InitAandB method calls SELF.ManipulateAandB without knowing which actual method is going to execute. When the Object1.InitAandB(10,20) statement executes, SELF.ManipulateAandB calls AnotherClass.ManipulateAandB, and when the Object2.InitAandB(30,40) statement executes, it calls the DifferentClass.ManipulateAandB method.

A virtual method in the derived class may explicitly call the method of the same name from the base class by calling PARENT.VirtualMethodName. This is just the same technique as previously demonstrated in “Overriding Inherited Methods.”

Local Derived Methods

Methods prototyped in a derived CLASS declaration within a procedure's Local data section share the declaring procedure's scope for all local data declarations and routines. The requirement for this is that the methods must be defined within the same source module as the declaring procedure and must immediately follow the procedure within that source. That is, the methods must come after any ROUTINEs and before any other procedures that may be in the same source module. This means the procedure's Local data declarations and ROUTINEs are all visible and can be referenced within these methods.

The most common need for this scoping is the definition of the VIRTUAL methods for a derived object declared locally to a procedure to serve some purpose for the declaring procedure. Typically, these virtual methods need access to the procedure's local data to perform their intended function.

For example:

```
                MEMBER('MyApp')          !A source module
SomeClass        CLASS,TYPE,MODULE('SomeClass.CLW')
PropertyA          LONG
PropertyB          LONG
InitAandB          PROCEDURE(LONG PassedA, LONG PassedB)
ManipulateAandB    PROCEDURE,VIRTUAL
                  END

MyProc          PROCEDURE               !Some non-object procedure
LocalVar        LONG                    !Local variable

AnotherClass    CLASS(SomeClass)        !Declare an object in local data
ManipulateAandB    PROCEDURE,VIRTUAL
                END
  CODE
  DO MyRoutine
  AnotherClass.InitAandB(10,20)         !Call base class method
MyRoutine  ROUTINE                      !Routine local to MyProc and
  LocalVar += 10                        ! AnotherClass.ManipulateAandB
AnotherClass.ManipulateAandB  PROCEDURE  !Virtual method with access to
X    LONG                                ! local data and routines
  CODE
  LOOP 10 TIMES
    X = (SELF.PropertyA * SELF.PropertyB * LocalVar)
    MESSAGE('The Product of A*B*LocalVar is ' & X)
    DO MyRoutine                         !Increment the local variable
  END
!MODULE('SomeClass.CLW') contains:
SomeClass.InitAandB  PROCEDURE(LONG PassedA, LONG PassedB)
  CODE
  SELF.PropertyA = PassedA
  SELF.PropertyB = PassedB
  SELF.ManipulateAandB              !Call whichever virtual method is
                                    ! appropriate for the current object
SomeClass.ManipulateAandB  PROCEDURE !Base class method does nothing
  CODE
```

Summary

- The three most important concepts in Object-orientation are: Encapsulation, Inheritance, and Polymorphism.
- The CLASS structure creates Encapsulation.
- The source code for all the methods in a single CLASS reside in a single source module, making maintenance easier.
- An object is an instance of a CLASS with its own set of data members (properties) which shares methods with all other instances of the CLASS (and also any derived CLASSes).
- A CLASS may contain data members (properties) declared as simple data types or as reference variables to complex data types (including other classes).
- Within the CLASS methods, the data members and methods appropriate to the current object instance are referenced using SELF as the object name (SELF.DataMemberName or SELF.MethodName).
- Constructor and destructor methods named Construct and Destruct are automatically called when an object is created or destroyed, and may also be explicitly called.
- All properties and methods are public unless explicitly declared with the PRIVATE or PROTECTED attribute.
- Inheritance is achieved by deriving one CLASS from another.
- You can override inherited methods by re-declaring and re-defining them in the derived CLASS, using exactly the same parameter list.
- Overridden inherited methods and VIRTUAL methods can call their base class constituents by using PARENT (just like SELF).
- Composition provides a viable alternative to multiple inheritance.
- VIRTUAL methods allow standard OOP Polymorphism, while Clarion's procedure overloading permits a non-OOP form of polymorphism.
- VIRTUAL methods are prototyped in both the base and derived classes.
- Objects local to a procedure share local variables and ROUTINEs with the declaring procedure.

4 - Database Design

Database Design

There are a number of methods of database organization in use today. The Inverted List Model, the Hierarchical Model, and the Network Model are three that have been widely used in the past. Mostly, these models have been used on mainframe computers, and have not been implemented on PC systems on a widespread basis. The Clarion language has the tools to allow you to utilize any of these methods, if you so choose.

By far, the most common method of database organization on PC systems today is the Relational Model, as defined by E. F. Codd. There is no database program which completely implements all of Codd's rules regarding relational database, because it is an extremely complex mathematical model. However, most database programs implement a sufficient sub-set of Codd's rules to allow practical use of the principles of the Relational Model. This essay is a very brief overview of the most fundamental aspects of relational database design as they impact business programming.

Relational Database Design

One basic principle of Relational Database involves the database design—a data item should be stored once—not duplicated in many places. There are two benefits to this: lowered disk space requirements, and easier data maintenance. To achieve this end, a relational database design splits the data into separate, related files. For example, assume a very simple order-entry system which needs to store the following data:

```
Customer Name
Customer Address
ShipTo Address
Order Date
Product Ordered
Quantity Ordered
Unit Price
```

This data could all be stored in each record of one file, but that would be very inefficient. The Customer Name, Address, ShipTo Address, and Order Date would be duplicated for every item ordered on every order. To eliminate the duplication, you split the data into separate files.

```
Customer File:   Customer Name
                 Customer Address
Order File:      ShipTo Address
                 Order Date
Item File:       Product Ordered
                 Quantity Ordered
                 Unit Price
```

With this file configuration, the Customer File contains all the customer information, the Order File contains all the information that is pertinent to one order, and the Item File contains all the information for each item in the order. This certainly eliminates duplicate data. However, how do you tell which record in what file relates to what other records in which other files? This is the purpose of the relational terms “Primary Key” and “Foreign Key.”

A Primary Key is an index into a file based on a field (or fields) that cannot contain duplicate or null values. To translate this to Clarion language terms: a Primary Key would be a unique KEY (no DUP attribute) with key components that are all REQuired fields for data entry. In strict relational database design, one Primary Key is required for every file.

A Foreign Key is an index into a file based on a field (or fields) which contain values that duplicate the values contained in the Primary Key fields of another, related, file. To re-state this, a Foreign Key contains a “reference” to the Primary Key of another file.

Primary Keys and Foreign Keys form the basis of file relationships in Relational Database. The matching values contained in the Primary and Foreign Keys are the “pointers” to the related records. The Foreign Key records in “File A” point back to the Primary Key record in “File B”, and the Primary Key in “File B” points to the Foreign Key records in “File A.”

Defining the Primary and Foreign Keys for the above example requires that you add some fields to the files to fulfill the relational requirements.

```
Customer File:  Customer Number - Primary Key
                Customer Name
                Customer Address

Order File:     Order Number - Primary Key
                Customer Number - Foreign Key
                ShipTo Address
                Order Date

Item File:      Order Number - 1st Primary Key Component and Foreign Key
                Product Ordered - 2nd Primary Key Component
                Quantity Ordered
                Unit Price
```

In the Customer File, there is no guarantee that there could not be duplicate Customer Names. Therefore, the Customer Number field is added to become the Primary Key. The Order Number has been added to the Order File as the Primary Key because there is no other field that is absolutely unique in that file. The Customer Number was also added as a Foreign Key to relate the Order File to the Customer File. The Item File now contains the Order Number as a Foreign Key to relate to the Order File. It also becomes the first component of the multiple component (Order Number, Product Ordered) Primary Key.

The Relational definitions of Primary Key and Foreign Key do not necessarily require the declaration of a Clarion KEY based on the Primary or Foreign Key. This means that, despite the fact that these Keys exist in theory, you will only declare a Clarion KEY if your application actually needs it for some specific file access. Generally speaking, most all Primary Keys will have a Clarion KEY, but fewer Foreign Keys need have Clarion KEYs declared.

File Relationships

There are three types of relationships that may be defined between any two files in a relational database: One-to-One; One-to-Many (also called Parent-Child) and its reverse view, Many-to-One; and Many-to-Many. These relationships refer to the number of records in one file that are related to some number of records in the second file.

In the previous example, the relationship between the Customer File and the Order File is One-to-Many. One Customer File record may be related to multiple Order File records. The Order File and the Item File also have a One-to-Many relationship, since one Order may have multiple Items. In business database applications, One-to-Many (Parent-Child) is the most common relationship between files.

A One-to-One relationship means that exactly one record in one file may be related to exactly one record in another file. This is useful in situations where a particular file may, or may not, need to have data in some fields. If all the fields are contained in one file, you can waste a lot of disk space with empty fields in those records that don't need the extra information. Therefore, you create a second file with a One-to-One relationship to the first file, to hold the possibly unnecessary fields.

To expand the previous example, an Order may, or may not, need to have a separate ShipTo Address. So, you could add a ShipTo File to the database design.

```
Order File:   Order Number - Primary Key
              Customer Number - Foreign Key
              Order Date

ShipTo File:  Order Number - Primary Key and Foreign Key
              ShipTo Address
```

In this example, a record would be added to the ShipTo File only if an Order has to be shipped to some address other than the address in the Customer File. The ShipTo File has a One-to-One relationship with the Order File.

Many-to-Many is the most difficult file relationship with which to deal. It means that multiple records in one file are related to multiple records in another file. Expand the previous example to fit a manufacturing concern which buys Parts and makes Products. One Part may be used in many different Products, and one Product could use many Parts.

```
Parts File:    Part Number - Primary Key
               Part Description
Product File:  Product Number - Primary Key
               Product Description
```

Without going into the theory, let me simply state that this situation is handled by defining a third file, commonly referred to as a “Join” file. This Join file creates two One-to-Many relationships, as in this example:

```
Parts File:    Part Number - Primary Key
               Part Description

Parts2Prod File:
               Part Number - 1st Primary Key Component and Foreign Key
               Product Number - 2nd Primary Key Component and Foreign Key
               Quantity Used

Product File:  Product Number - Primary Key
               Product Description
```

The Parts2Prod File has a multiple component Primary Key and two Foreign Keys. The relationship between Parts and Parts2Prod is One-to-Many, and the relationship between Product and Parts2Prod is also One-to-Many. This makes the Join file the “middle-man” between two files with a Many-to-Many relationship.

An advantage of using a Join file is that there is usually some more information that logically should be stored there. In this case, the Quantity Used (of a Part in a Product) logically only belongs in the Parts2Prod file.

Translating the Theory to Clarion

In practical relational database design, a Clarion KEY may not need to be declared for the Primary Key on some files. If there is never a need to directly access individual records from that file, then a KEY definition based on the Primary Key is not necessary. Usually, this would be the Child file (of a Parent-Child relationship) whose records are only needed in conjunction with the Parent record.

A Clarion KEY also may not need to be declared for a Foreign Key. The determination to declare a KEY is dependent upon how you are going to access the file containing the Foreign Key. If you need to access the Foreign Key records from the Primary Key, a Clarion KEY is necessary. However, if the only purpose of the Foreign Key is to ensure that the value in the Foreign Key field value is valid, no Clarion KEY is needed. Take the previous theoretical examples and create Clarion file definitions:

```
Customer    FILE,DRIVER('Clarion'),PRE(Cus)
CustKey      KEY(Cus:CustNo)          !Primary KEY
Record       RECORD
CustNo        LONG                  !Customer Number - Primary Key
Name          STRING(30)            !Customer Name
Address       STRING(30)            !Customer Address
            . .

Order         FILE,DRIVER('Clarion'),PRE(Ord)
OrderKey      KEY(Ord:OrderNo)        !Primary KEY
CustKey       KEY(Ord:CustNo),DUP     !Foreign KEY
Record         RECORD
OrderNo         LONG                  !Order Number - Primary Key
CustNo          LONG                  !Customer Number - Foreign Key
Date            LONG                  !Order Date
               . .

ShipTo       FILE,DRIVER('Clarion'),PRE(Shp)
OrderKey      KEY(Shp:OrderNo)        !Primary KEY
Record        RECORD
OrderNo        LONG          !Order Number - Primary Key and Foreign Key
Address        STRING(30)    !ShipTo Address
             . .

Item         FILE,DRIVER('Clarion'),PRE(Itm)
OrderKey      KEY(Itm:OrderNo,Itm:ProdNo)  !Primary KEY
Record        RECORD
OrderNo        LONG            !Order - Primary Component and Foreign Key
ProdNo         LONG            !Prod. - Primary Component and Foreign Key
Quantity       SHORT           !Quantity Ordered
Price          DECIMAL(7,2)    !Unit Price
             . .
```

```
Product       FILE,DRIVER('Clarion'),PRE(Pro)
ProdKey        KEY(Pro:ProdNo)          !Primary KEY
Record          RECORD
ProdNo           LONG                  !Product Number - Primary Key
Description      STRING(30)            !Product Description
              . .

Parts2Prod  FILE,DRIVER('Clarion'),PRE(P2P)
ProdPartKey  KEY(P2P:ProdNo,P2P:PartNo)  !Primary KEY
PartProdKey  KEY(P2P:PartNo,P2P:ProdNo)  !Alternate KEY
Record        RECORD
PartNo         LONG            !Part - Primary Component and Foreign Key
ProdNo         LONG            !Prod. - Primary Component and Foreign Key
Quantity       SHORT
            . .

Parts         FILE,DRIVER('Clarion'),PRE(Par)
PartKey        KEY(Par:PartNo)              !Primary KEY
Record          RECORD
PartNo           LONG                     !Part Number - Primary Key
Description      STRING(30)               !Part Description
              . .
```

Notice that only one Foreign Key (in the Order file) was explicitly declared as a Clarion KEY. A number of Foreign Keys were included as part of Primary Key declarations, but this was simply good fortune.

The Primary Key (Itm:OrderKey) defined on the Item file is there to ensure that an order does not contain duplicate Products Ordered. If this were not a consideration, Itm:OrderKey would only contain Itm:OrderNo, and would have the DUP attribute to allow duplicate KEY values. This would make it a Foreign Key instead of a Primary Key, and the file would not have a KEY defined for the Primary Key.

The Item file and the Product file have a Many-to-One relationship, which is One-to-Many looked at from the reverse perspective. This reverse view is most often used for data entry verification look-up. This means the Product Number entered into the Item file's data entry procedure can look-up and verify the Product Number against the records in the Product file.

Referential Integrity

There is one more fundamental issue in the Relational Model which should be addressed: "Referential Integrity." This is an issue which must be resolved in the executable source code for an application, because it involves the active, run-time inter-relationship of the data within the database.

Referential Integrity means that no Foreign Key can contain a value that is not matched by some Primary Key value. Maintaining Referential Integrity in your database begets two questions which must be resolved:

- What do you do when the user wants to delete the Primary Key record?

- What do you do when the user wants to change the Primary Key value?

The three most common answers to each of these questions are: Restrict the action, Cascade the action, or (less commonly) Nullify the Foreign Key values. Of course, there may also be application-specific answers, such as copying all information to history files before performing the action, which should be implemented as required in individual programs.

Restrict the action

Restrict the action means that when the user attempts to delete the Primary Key record, or change the Primary Key value, the action is only allowed if there are no Foreign Keys that reference that Primary Key. If related Foreign Keys do exist, the action is not allowed.

Using the files defined previously, here is an example of how the executable code might look to Restrict deletes or a change of the Primary Key value.

```
ChangeRec  EQUATE(2)      !EQUATE Change Action
DeleteRec  EQUATE(3)      !EQUATE Delete Action value for readability
SaveKey    LONG          !Primary Key save variable
  CODE
  SaveKey = Cus:CustNo               !Save Primary Key value
  OPEN(window)
  ACCEPT
    CASE ACCEPTED()                  !Process entry
        !individual control processing
    OF ?OKButton                     !Screen completion button
      IF Action = ChangeRec AND Cus:CustNo <> SaveKey
                                     !Check for changed Primary Key value
        Cus:CustNo = SaveKey         ! change it back
        MESSAGE('Key Field changes not allowed!')    !tell the user
        SELECT(1)                    !  to start over
        CYCLE
      ELSIF Action = DeleteRec       !Check for Delete Action
        Ord:CustNo = Cus:CustNo      !Initialize Key field
        GET(Order,Ord:CustKey)       !  and try to get a related record
        IF NOT ERRORCODE()           !If the GET was successful
          MESSAGE('Delete not allowed!') ! tell user
          SELECT(1)                      !  to start over
          CYCLE
        ELSE                             !If GET was unsuccessful
          DELETE(Customer)               !go ahead and delete it
          BREAK                          !and get out
        END
        !other executable processing statements
      END
    END
  END
```

Cascade the action

Cascade the action means that when the user attempts to delete the Primary Key record, or change the Primary Key value, the action cascades to include any Foreign Keys that reference that Primary Key. If related Foreign Keys do exist, the delete action also deletes those records, and the change action also changes the values in the Foreign Keys that reference that Primary Key.

There is one consideration that should be noted when you Cascade the action. What if the file you Cascade to (the Child file) is also the Parent of another Child file? This is a situation which you must detect and handle, because the Cascade action should affect all the dependent file records. When you are writing source code to handle this situation, you need to be aware of the file relationships and write code that Cascades the action as far it needs to go to ensure that nothing is "left hanging."

Again using the files defined previously, here is an example of how the executable code might look to Cascade deletes or a change of the Primary Key value.

```
ChangeRec  EQUATE(2)      !EQUATE Change Action
DeleteRec  EQUATE(3)      !EQUATE Delete Action value for readability
SaveKey    LONG           !Primary Key save variable
  CODE
  SaveKey = Cus:CustNo    !Save Primary Key value
  OPEN(window)
   ACCEPT
    CASE ACCEPTED()       !Process entry
        !individual control processing
    OF ?OKButton          !Screen completion button
      IF Action = ChangeRec AND Cus:CustNo <> SaveKey
                          !Check for changed Primary Key value
        DO ChangeCascade                !  and cascade the change
      ELSIF Action = DeleteRec          !Check for Delete Action
        DO DeleteCascade                !  and cascade the delete
      END
        !other executable processing statements
    END
  END

ChangeCascade  ROUTINE
  Ord:CustNo = SaveKey                !Initialize the key field
  SET(Ord:CustKey,Ord:CustKey)        ! and set to process all of one
  LOOP                                !   customer's orders
    NEXT(Order)                       !    one at a time
    IF Ord:CustNo <> SaveKey OR ERRORCODE() THEN BREAK.
                                      !Check for end of cust. and get out
    Ord:CustNo = Cus:CustNo           !Change to new value
    PUT(Order)                        !  and put the record back
    IF ERRORCODE() THEN STOP(ERROR()).
  END

DeleteCascade  ROUTINE
  Ord:CustNo = SaveKey                !Initialize the key field
  SET(Ord:CustKey,Ord:CustKey)        ! and set to process all of one
```

```
LOOP                                !   customer's orders
  NEXT(Order)                       !    one at a time
  IF Ord:CustNo <> SaveKey OR ERRORCODE() THEN BREAK.
                                    !Check for end of cust. and get out
  CLEAR(Itm:Record)                 !Clear the record buffer
  Itm:OrderNo = Ord:OrderNo         !Initialize the key field
  SET(Itm:OrderKey,Itm:OrderKey)    ! and set to process all of one
  LOOP UNTIL EOF(Item)              !   order's items
    NEXT(Item)                      !    one at a time
    IF Itm:OrderNo <> Ord:OrderNo OR ERRORCODE() THEN BREAK.
                      !Check for end of order and get out of Item loop
    DELETE(Item)                    !  and delete the Item record
    IF ERRORCODE() THEN STOP(ERROR()).
  END                               !End Item file loop
  Shp:OrderNo = Ord:OrderNo         !Check for ShipTo record
  GET(ShipTo,Shp:OrderKey)
  IF NOT ERRORCODE()                !If GET was successful
    DELETE(ShipTo)                  !  delete the ShipTo record
    IF ERRORCODE() THEN STOP(ERROR()).
  END
  DELETE(Order)                     !  and delete the Order record
  IF ERRORCODE() THEN STOP(ERROR()).
END                                 !End Order file loop
```

Nullify the Foreign Key

Nullify the Foreign Key means that when the user attempts to delete the Primary Key record, or change the Primary Key value, the Foreign Keys that reference that Primary Key are changed to null values (if the Foreign Key fields allow null values).

Again using the files defined previously, here is an example of how the executable code would look to Nullify the Foreign Keys on delete or a change of the Primary Key value.

```
ChangeRec  EQUATE(2)      !EQUATE Change Action
DeleteRec  EQUATE(3)      !EQUATE Delete Action value for readability
SaveKey    LONG           !Primary Key save variable
  CODE
  SaveKey = Cus:CustNo               !Save Primary Key value
  OPEN(window)
   ACCEPT
    CASE ACCEPTED()                  !Process entry
        !individual control processing
    OF ?OKButton                     !Screen completion button
      IF Action = ChangeRec AND Cus:CustNo <> SaveKey
                                     !Check for changed Primary Key value
        DO ChangeNullify             !  and nullify the Child records
      ELSIF Action = DeleteRec       !Check for Delete Action
        DO DeleteNullify             !  and nullify the Child records
      END
        !other executable processing statements
    END
  END

ChangeNullify  ROUTINE
  Ord:CustNo = SaveKey               !Initialize the key field
  SET(Ord:CustKey,Ord:CustKey)       ! and set to process all of one
  LOOP                               !   customer's orders
    NEXT(Order)                      !    one at a time
    IF Ord:CustNo <> SaveKey OR ERRORCODE() THEN BREAK.
                                     !Check for end of cust. and get out
    Ord:CustNo = 0                   !Change to null value
    PUT(Order)                       !  and put the record back
    IF ERRORCODE() THEN STOP(ERROR()).
  END

DeleteNullify  ROUTINE
  Ord:CustNo = SaveKey               !Initialize the key field
  SET(Ord:CustKey,Ord:CustKey)       ! and set to process all of one
  LOOP                               !   customer's orders
    NEXT(Order)                      !    one at a time
    IF Ord:CustNo <> SaveKey OR ERRORCODE() THEN BREAK.
                                     !Check for end of cust. and get out
    Ord:CustNo = 0                   !Change to null value
    PUT(Order)                       !  and put the record back
    IF ERRORCODE() THEN STOP(ERROR()).
  END
```

The Nullify option does not require as many changes as the Cascade option. This is because the Cascade has to delete all the related records in as many files as are related. Nullify only needs to null out the individual Foreign Keys that reference the Primary Key being changed or deleted.

Summary

- Each data item should be stored once.
- Separate files are used to eliminate data duplication.
- Files are related by Primary and Foreign Keys.
- A Primary Key is a unique (and non-null) index into a file which provides for individual record access.
- A Foreign Key contains a reference to the Primary Key of some other file.
- One-to-Many file relationships are the most common. They are also referred to as Parent-Child and Many-to-One (same relationship, reverse view).
- One-to-One file relationships are most commonly created to hold data that is not always needed in every record.
- Many-to-Many relationships require a “Join” file which acts as a broker between the two files. The Join file inserts two One-to-Many relationships between the Many-to-Many relationship.
- Only those Primary and Foreign Keys that the application needs (as a practical consideration) for specific access to the files need to have Clarion KEYs declared.
- Referential Integrity means that all Foreign Keys contain valid references to Primary Keys.
- Maintaining Referential Integrity requires executable code that tests for Update or Delete of the Primary Key values.
- The three common solutions to maintaining Referential Integrity are: Restricting (update/delete not allowed), Cascading (also update/delete the Foreign Key), or Nullifying the Foreign Key (assign null values to the Foreign Key).

5 - Data File Processing

Data File Processing

Custom database applications, by definition, store data in files. Getting data into those files, and processing it for some kind of meaningful output, is the primary purpose of any database application. This essay is a discussion of the Clarion language tools that allow the programmer to access and process data files.

File Access Methods

Generally speaking, records are put into data files at the end of the file in the sequence in which they are added (this is not always true, but is usually true). This creates the “physical, record-number order” of the file—the physical order in which the records appear within the file. This physical order does not necessarily correspond to any meaningful or useful sequence.

There are two ways to access records within a file: sequential access, and random access. Sequential access means you retrieve a number of records in some specified sequence, processing each record in order. Random access means you retrieve and process one specific record. Both of these access methods are used in almost every business database application.

If you only need to access records sequentially in their physical, record-number order, nothing more than the data file is needed. If you need to randomly access a record, and you know exactly which position it occupies in the file (its record number), the same thing is true. However, for most applications, these constraints would be too limiting.

KEY and INDEX

The Clarion KEY and INDEX declarations create alternate sort orders for the records in the file. These allow sequential or random access to a data file in some order other than the physical, record-number order. The order is determined by the component fields that make up the KEY or INDEX. Each KEY or INDEX component may be in ascending or descending order.

The main difference between KEY and INDEX lies in the fact that a KEY is dynamically maintained. Every time a record is added, changed, or deleted, the KEY is also updated. Since it is always kept current, a KEY should be used for sort orders that are frequently used in the application.

An INDEX is not maintained and must be rebuilt immediately before it is used to ensure that it accurately reflects the current state of the file. The BUILD statement is used to rebuild an INDEX. Because of the time factor in rebuilding, and the fact that exclusive file access is required for the BUILD, an INDEX should be used for sort orders that are infrequently used.

One special form of INDEX is the ”dynamic” INDEX. This is an INDEX whose component fields are not declared in the file definition. The component fields of a “dynamic” INDEX are declared at run-time in the BUILD statement.

Unlike a “static” INDEX, you may BUILD a “dynamic” INDEX with the file open in any access mode. The advantage should be immediately obvious—end-user-definable sort orders.

```
Sample     FILE,DRIVER('TopSpeed'),PRE(Sam)
Field1Key   KEY(Sam:Field1)                !KEY on Field 1
Field2Ndx   INDEX(Sam:Field2)              !Static INDEX on Field2
DynNdx      INDEX()                        !Dynamic INDEX
Record       RECORD
Field1         LONG
Field2         STRING(10)
Field3         DECIMAL(7,2)
           . .

   CODE
   OPEN(Sample,42h)                        !Open read/write deny none
   LOCK(Sample)                            !Lock for exclusive access
   BUILD(Sam:Field2                        !  to Build the INDEX
   UNLOCK(Sample)                          !    then Unlock the file
   BUILD(Sam:DynNdx,'-Sam:Field1,+Sam:Field2')   !Build the dynamic INDEX
```

In this example, the KEY on Sam:Field1 will always be current, the INDEX on Sam:Field2 is built when the file is opened and LOCKed (exclusive access is required). The "dynamic" INDEX is built at run-time in descending Sam:Field1 and ascending Sam:Field2 sort order. Of course, it could be built in any sort order possible for the file.

Other than their maintenance, KEY and INDEX are functionally equivalent. They share the same type of file format and may be used interchangeably in all executable file access statements which require a KEY or INDEX parameter. To simplify this discussion, wherever the phrase "KEY and/or INDEX" would be appropriate, it is replaced with the generic term index. All references to index apply equally to both KEYs and INDEXes, unless otherwise noted.

Sequential File Access

There are three Clarion statements which perform sequential file access: SET, NEXT, and PREVIOUS. The SET statement initializes the sequential processing—it does not read a record. The NEXT and PREVIOUS statements read the records in ascending (NEXT) or descending (PREVIOUS) order within the sequence established by SET.

The SET statement is the ruling element in sequential file processing. A SET statement must come before NEXT or PREVIOUS to initialize the starting point and sequence in which the records will be read. Usually, the SET statement is the last executable statement before the LOOP structure which sequentially processes the records in the file. The NEXT or PREVIOUS is then the first statement within the LOOP, as in this example code using the previous example file definition:

```
SET(Sam:Field1Key)              !Set to top of file in KEY order
LOOP                            !Loop until end of the Sample file
    NEXT(Sample)                !Read each record in turn
 IF ERRORCODE() THEN BREAK.     !Break at end of file
       !record processing statements
```

```
    END                                 !End loop
```

There are seven forms of the SET statement listed in the Language Reference Manual. These essentially break down into two categories: three starting points for physical record-number order access, and four starting points for indexed order access.

Physical Order	Indexed Order
Top/Bottom of File	Top/Bottom of File
Physical Record Number	Index Record Number
Index Value	Index Value
	Index Value and Physical Record Number

SET initializes the sequential processing record pointer, and it employs a type of "fuzzy logic." When you SET to the Top/Bottom of the file, the record pointer is not actually pointing at either. If you issue a NEXT after the SET, you read records forward from the beginning of the file. If you issue a PREVIOUS instead, you read records backwards from the end of the file. Once you have issued the NEXT or PREVIOUS to begin reading records in one direction, you cannot go back across the Top/Bottom of the file without another SET.

The same "fuzzy logic" is active when you SET to an index value. If SET finds a record containing an exact match to that index value, it points to that specific record. In this case, either NEXT or PREVIOUS would read the same record.

If, however, there is no exact match to the index value, SET points "between" the last record in sequence containing a value less than (or greater than, in a descending index) the index value and the next record in sequence containing a value greater than (or less than, in a descending index) the index value. In this case, NEXT and PREVIOUS would not read the same record. NEXT would read the following record in the index sequence, PREVIOUS would read the prior record in the index sequence.

The advantage of this "fuzzy logic" lies in its use with a multiple component index, as in this example.

```
Sample     FILE,DRIVER('TopSpeed'),PRE(Sam)
FieldsKey   KEY(Sam:Field1,Sam:Field2),DUP   !KEY on Field 1 and Field 2
Record       RECORD
Field1        LONG
Field2        STRING(10)
Field3        DECIMAL(7,2)
           . .
   CODE
   OPEN(Sample,42h)                    !Open read/write deny none
   CLEAR(Sam:Record)                   !Clear the record buffer
   Sam:Field1 = 10                     !Initialize first KEY component
   SET(Sam:FieldsKey,Sam:FieldsKey)    !KEY sequence, start at 10-blank
   LOOP                                !Process each record
       NEXT(Sample)                    !    one at a time
    IF ERRORCODE() THEN BREAK.         !Break at end of file
       IF Sam:Field1 <> 10             !Check for end of group
           BREAK                       !  if so, get out of process loop
       END
      !record processing statements
   END                                 !End process loop
```

This code first clears the record buffer, assigning zeroes to Sam:Field1 and Sam:Field3, and blanks to Sam:Field2. The first component field of Sam:FieldsKey is initialized to the value that must be in the records you need to process. The SET statement sets up sequential processing in indexed order, starting at the index value—in this case a value of 10 in Sam:Field1 and blanks in Sam:Field2.

Sample File Records:	Index Record #	Field1	Field2	Field3
	1	5	ABC	14.52
	2	5	DEF	14.52
Record Pointer After SET >>				
	3	10	ABC	14.52
	4	10	ABC	29.04
	5	10	DEF	14.52
	6	15	ABC	14.52
	7	15	DEF	14.52

SET leaves the record pointer positioned as shown above because there is no exact match. Record 2's value 5-DEF is less than 10-blank, and record 3's value 10-ABC is greater than 10-blank, therefore the record pointer is left "between" the two. The first time through the LOOP, NEXT reads record number 3. The IF statement terminates the processing loop after NEXT reads record 6.

There is a distinct difference between the Physical Record Number and the Index Record Number. The Physical Record Number is the relative physical position within the data file as returned by the POINTER(Label of a File) procedure. The Index Record Number is the relative record position within the index sequence as returned by the POINTER(Label of an Index) procedure.

In physical order, the same file might look like this (of course, the physical and index record numbers are not stored in the data file):

Sample File:	Physical Record #	Index Record #	Field1	Field2	Field3
	1	3	10	ABC	14.52
	2	6	15	ABC	14.52
	3	5	10	DEF	14.52
	4	2	5	DEF	14.52
	5	4	10	ABC	29.04
	6	7	15	DEF	14.52
	7	1	5	ABC	14.52

The forms of SET that use Record Numbers as the starting point look very similar, therefore you need to be very clear about which you are using (Physical vs. Index).

```
SET(Sample,1)        !Physical Order, SETs to physical rec 1, index rec 3

SET(Sam:FieldsKey,1)
                     !Index order, SETs to index rec 1, physical rec 7

Sam:Field1 = 10
Sam:Field2 = 'ABC'
SET(Sam:FieldsKey,Sam:FieldsKey,5)
                     !Index order, SETs to index rec 4, physical rec 5
```

This last form of SET allows you to SET to a specific record within a sequence of records which contain duplicate index field values. It searches the duplicate index entries for an index entry which points to the Physical Record Number specified as the third parameter. This is useful in files where there are multiple records with duplicate index values and you need to begin processing at one specific record within those duplicates.

Random File Access

There is only one Clarion statement which performs random access to individual records within a file—the GET statement. Unlike SET, GET either reads the record you attempt to retrieve, or returns an error. There is no "fuzzy logic" with GET.

There are three forms of the GET statement. They allow you to retrieve a record based on an index value, Physical Record Number, or Index Record Number.

```
Sam:Field1 = 15
Sam:Field2 = 'ABC'
GET(Sample,Sam:FieldsKey)         !GETs index rec 6, physical rec 2
GET(Sample,1)                     !GETs physical rec 1, index rec 3
GET(Sam:FieldsKey,1)              !GETs index rec 1, physical rec 7
```

The first GET example retrieves the first record in the index order that contains the values in the index component fields at the time the GET is issued. The second example retrieves the first record in the file in Physical Record Number order. The third retrieves the first record in the file in Index Record Number order.

GET always looks for an exact match to the index value and returns an error if it does not find one. Therefore, all component fields of a multiple component index must be initialized before issuing a GET.

GET is completely independent of SET/NEXT or SET/PREVIOUS sequential processing. This means that a GET into a file which is being sequentially processed does not change the record pointer for sequential processing.

```
SET(Sam:FieldsKey)                  !Set to top of file
LOOP                                !Process each record in index order
  NEXT(Sample)                      !Gets each sequential record
  IF ERRORCODE() THEN BREAK.        !Break at end of file
     !sequential record processing statements
  GET(Sam:FieldsKey,1)              !Gets the first record in index order
     !random access record processing statements
END
```

This example code processes through the entire file in index order. After each record is processed, the first record in index order is retrieved and processed. This does not affect the sequence, therefore NEXT will progress through the file, despite the GET of the first record every time through the loop.

Summary

- Sequential Access and Random Access are the two methods used to retrieve records from a file.
- The Clarion KEY and INDEX declarations define alternate sort orders of the file in which they are declared.
- A KEY is dynamically maintained and is always ready for use. An INDEX is not maintained and must be built before use.
- A “dynamic” INDEX allows sort orders to be defined at run-time.
- The SET statement initializes the order and starting point of sequential processing. A SET is required before the first NEXT or PREVIOUS.
- SET employs “fuzzy logic” to determine the starting point. It either points to a specific record, or “between” records at the position where it determined no record fit the starting point parameters it was given.
- Physical and Index Record Numbers are very different and must not be confused with each other.
- The GET statement performs random record access within a file.
- GET is completely independent of the SET/NEXT and SET/PREVIOUS sequential record processing.

6 - Multi-User Considerations

Multi-User Considerations

The world of database applications programming is rapidly heading towards networking. Stand-alone applications are expanding into multi-user environments as more companies connect their PCs to Local Area Networks (LANs). Mainframe applications in large companies are being “right-sized” and re-written for LAN operation. With the emergence of multi-threading, multi-tasking operating systems for PCs, even standalone computers need applications that are written with multi-user shared-access considerations in mind. This essay is a discussion of the Clarion language tools provided to write applications specifically designed for use in multi-user environments.

Opening Files

Before any data file can be processed, it must first be opened. The OPEN and SHARE statements provide this function. OPEN and SHARE are functionally equivalent, the only difference between the two is the default value of the second (access mode) parameter of each.

The access mode specifies the type of access the user opening the file receives, and the type of access allowed to other users of the file. These two values are added together to create the DOS Access Code for the file. The access mode values are:

	Access	Dec.	Hex.
User’s Access:	Read Only	0	0h
	Write Only	1	1h
	Read/Write	2	2h
Other’s Access:	Deny All	16	10h
	Deny Write	32	20h
	Deny Read	48	30h
	Deny None	64	40h

The OPEN statement’s default access mode is Read/Write Deny Write (22h), which only allows exclusive (single-user) disk write access to the user opening the file. The SHARE statement’s default access mode is Read/Write Deny None (42h), allowing non-exclusive (multi-user) access to anybody who opens the file. Either OPEN or SHARE may open the file in any of the possible access modes.

```
OPEN(file)            !Open Read/Write Deny Write
OPEN(file,22h)        !Open Read/Write Deny Write
SHARE(file,22h)       !Open Read/Write Deny Write
SHARE(file)           !Open Read/Write Deny None
SHARE(file,42h)       !Open Read/Write Deny None
OPEN(file,42h)        !Open Read/Write Deny None
OPEN(file,40h)        !Open Read Only Deny None
SHARE(file,40h)       !Open Read Only Deny None
```

These examples demonstrate the three most commonly used access modes. For multi-user applications, the most common access mode is Read/Write Deny None (42h), which permits all users complete access to the file. Read Only Deny None (40h) is usually used in multi-user situations where the user will not update the file (a lookup-only file) but there may be some other user who may need to write to that file.

Concurrency Checking

The biggest consideration to keep in mind about multi-user access to files is the possibility that several users could be updating the same record at the same time. A process known as "concurrency checking" prevents the data file from being corrupted by multiple user updates to the same record. Concurrency checking means determining that the record on disk, which is about to be overwritten, still contains the same values it did when it was retrieved for update.

Obviously, there is no need for any kind of concurrency checking when a record is being added. If the file has a unique KEY, two users adding the same record twice is impossible because the second ADD returns a "Creates Duplicate Key" error without adding the record. If duplicate KEYs are allowed, there is no generic way for the program code to check for inadvertent (incorrect) duplicates as opposed to deliberate (correct) duplicate records. There is also no need for concurrency checking when a record is being deleted. Once the first user has deleted the record, it is gone. Any subsequent user that attempts to delete that record will not be able to get it in the first place.

Concurrency checking is necessary when a user is making a change to a record. The process of changing a record is: get the record, make the changes, and write the changes back to the file. The problem is, during the time it takes the first user to make changes to the record, a second user (a faster typist) could: get the same record, make some change, and write the changed record back to disk. When the time comes for the first user to write his/her changes to disk, the record on disk is no longer the same as when it was first retrieved. Does the first user simply overwrite the second (faster) user's changes? If both users are changing different data elements within that record and both changes are valid, overwriting the second user's changes cannot be allowed. Even if they are both making the same change, the first user needs to know that someone else has already made that change.

The simplest concurrency checking method is to execute the WATCH statement just before getting the record from disk. This tells the file driver to automatically perform concurrency checking and report an error on the PUT statement if there is a conflict. Unfortunately, not all file drivers support this.

For the simplest concurrency checking method without using WATCH, your program code should:

1 Save a copy of the record before any changes are made.

2 Re-read the record immediately before writing the changes to disk, and compare it with the saved original.

3 If the two are the same, allow the user's changes to be written to disk. If not, alert the user and display the record, as changed by the other user.

Assume the following global declarations and compiler equates:

```
Sample     FILE,DRIVER('TopSpeed'),PRE(Sam)      !A data file declaration
Field1Key   KEY(Sam:Field1)
Record      RECORD
Field1       LONG
Field2       STRING(10)
           . .
Action     LONG                            !Record update action variable
AddRec     EQUATE(1)
ChangeRec  EQUATE(2)
```

Assume that some procedure allows the user to select a record from the file, defines the expected file Action (Add, Change, or Delete the record), then calls an update procedure. The update procedure operates only on that selected record and accomplishes the Action the user set in the previous procedure. The update procedure's logic would be something like this:

```
Update      PROCEDURE                       !An update procedure
Screen      WINDOW
               !data entry screen declarations go here
            END
SaveQue     QUEUE,PRE(Sav)                  !Record save queue is a copy
SaveRecord      LIKE(Sam:Record),PRE(Sav)   !of the file's record buffer
            END                             !with a different prefix
SavRecPos   STRING(512)                     !Record position save variable
   CODE
   OPEN(Screen)
   Sav:SaveRecord = Sam:Record              !Save copy of record
   ADD(SaveQue,1)                           !  to QUEUE entry 1
   SavRecPos = POSITION(Sample)             !Save record position
   DISPLAY                                  !Display the record on screen
   ACCEPT                                   !Screen field process loop
       CASE ACCEPTED()
          !Individual screen field edit code goes here
       OF ?OKButton                         !Screen completion field
           IF Action = ChangeRec            !If changing an existing record
               Sav:SaveRecord = Sam:Record  !Save changes made
               ADD(SaveQue,2)               !  to QUEUE entry 2
               GET(SaveQue,1)               !Get original record from QUEUE
               REGET(Sample,SavRecPos)      !Get record from FILE again
               IF ERRORCODE()
                IF ERROR() = 'RECORD NOT FOUND' !Did someone else delete it?
                 Action = AddRec            ! change Action to add it back
                 GET(SaveQue,2)             !Get this user's changes
                 Sam:Record = Sav:SaveRecord! put them in record buffer
                ELSE
                 STOP(ERROR())              !Stop on any other error
                END
               ELSIF Sav:SaveRecord <> Sam:Record !Compare for other's changes
                Sav:SaveRecord = Sam:Record ! Save new disk record
                ADD(SaveQue,1)              !    to QUEUE entry 1
                DISPLAY                     ! Display other's changes
                BEEP                        ! Alert the user
                MESSAGE('Changed by another station')
                SELECT(1)                   !   and start over
```

```
                    CYCLE                          !    at first field
                   ELSE                            !If nobody changed it
                    GET(SaveQue,2)                 ! Get this user's changes
                    Sam:Record = Sav:SaveRecord    ! put them back in record buffer
                  . .
                EXECUTE Action                     !Execute disk write
                 ADD(Sample)                       !If Action = 1 (AddRec)
                 PUT(Sample)                       !If Action = 2 (ChangeRec)
                 DELETE(Sample)                    !If Action = 3 (DeleteRec)
                END
                ErrorCheck                         !A generic error checking procedure
                FREE(SaveQue)                      !Free memory used by queue entries
                BREAK                              !and break out of process loop
        . .                                        !End loop and case
```

This example code demonstrates the simplest type of concurrency checking you can do without using WATCH. It saves the original record in memory QUEUE entry one, and the position of that record in a STRING variable. After that, the user is allowed to make the changes to the screen data. The code to check for other user's changes is contained in the CASE FIELD() OF ?OKButton. This would be the field which the user completes when he/she is done making changes and is ready to write the record to disk.

To check for other user's changes, the code first saves this user's changes to a second memory QUEUE entry, then gets the saved original record from the QUEUE. The saved record position is used to get the record from the data file again. If the record is not found in the file, someone else has deleted it. Therefore, since this user is changing it, simply add the changed record back into the file. If the record was not deleted, it is compared against the original saved copy. If they are not the same, the changed record is saved to the same memory QUEUE entry (one) which contained the original record. Then the user is alerted to the problem and sent back to the first field on the screen to re-enter the changes (if necessary). If the record is still the same, the user's changes are retrieved from the second memory QUEUE entry and put into the record buffer for the disk write. This method is fairly straight-forward and logical. However, it uses three extra chunks of memory the size of the record buffer: the memory QUEUE's buffer, and the two entries in that QUEUE (plus each QUEUE entry's 28-byte overhead). If you are dealing with a file that has many fields, the record buffer could be very large and this could use a significant amount of memory.

Another method of concurrency checking does not copy and save the original record, but instead calculates a Checksum or Cyclical Redundancy Check (CRC) value. The calculation is performed on the record before changes are made, then the record is retrieved from disk and the calculation is performed again. If the two values are not the same, the record has been changed. This method still requires a save area for the user's changes, because the record must be read again for the second calculation, and all disk reads are placed in the record buffer. Without a save area, the user's changes would be overwritten.

Here is an example of a 16-bit CRC procedure, and its prototype for the MAP structure. This is similar to the CRC calculations used in some serial communications protocols. An array of BYTE fields is passed to the procedure, it calculates a 16-bit CRC value for that array, and returns it to a USHORT (16-bit unsigned) variable.

```
   MAP                              !The procedure prototype for the MAP.
       CRC16(*BYTE[]),USHORT        !CRC16 expects an array of BYTEs to be
   END                              !  passed to it, returns a USHORT value
!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
CRC16      PROCEDURE(Array)         !16 Bit CRC Check
CRC        ULONG                    !Work variable
   CODE
   LOOP X# = 1 TO MAXIMUM(Array,1)!Loop through whole array
       CRC = BOR(CRC,Array[X#])    !Concatenate an array byte to CRC
       LOOP 8 TIMES                !Loop through each bit
           CRC = BSHIFT(CRC,1)     !Shift CRC left one bit
           IF BAND(CRC,1000000h)   !Was CRC 24th bit on before shift?
            CRC = BXOR(CRC,102100h) ! XOR shifted value with CRC mask
   .   .   .    !End both loops
   RETURN(BAND(BSHIFT(CRC,-8),0000FFFFh)) !Shift and mask return value
```

Using this CRC check procedure, the previous example code would be changed to look like:

```
Update       PROCEDURE                        !An update procedure
Screen       WINDOW
                !data entry screen declarations go here
             END
Sav:SaveRecord     LIKE(Sam:Record),PRE(Sav),STATIC
                                    !Record buffer save area
                                    !with a different prefix
PassArray     BYTE,DIM(SIZE(Sam:Record),OVER(Sam:Record)
                                    !Declare array OVER Sam:Record
SavRecPos     STRING(512)           !Record position save variable
SavCRC       USHORT                 !CRC value save variable
   CODE
   OPEN(Screen)
   SavCRC = CRC16(PassArray)        !Save original CRC value
   SavRecPos = POSITION(Sample)     !Save record position
   DISPLAY                          !Display the record on screen
   ACCEPT                           !Screen field process loop
       CASE ACCEPTED()
        !Individual screen field edit code goes here
       OF ?OKButton                 !Screen completion field
           IF Action = ChangeRec    !If changing an existing record
               Sav:SaveRecord = Sam:Record   !Save changes made
               REGET(Sample,SavRecPos)       !Get record from FILE again
               IF ERRORCODE()
                IF ERROR() = 'RECORD NOT FOUND' !Did someone else delete it?
                 Action = AddRec               ! change Action to add it back
                 Sam:Record = Sav:SaveRecord   ! put them in record buffer
                ELSE
                 STOP(ERROR())                 !Stop on any other error
                END
               ELSIF SavCRC <> CRC16(PassArray) !Compare CRCs for changes
                SavCRC = CRC16(PassArray)       ! Save new CRC value
                DISPLAY                         ! Display other's changes
                BEEP                            ! Alert the user
                IF MESSAGE('Changed by another station').
                 SELECT(1)                      !   and start over
                 CYCLE                          !    at first field
                ELSE                            !If nobody changed it
                 Sam:Record = Sav:SaveRecord    ! put changes back in buffer
              . .
           EXECUTE Action                       !Execute disk write
            ADD(Sample)                         !If Action = 1 (AddRec)
            PUT(Sample)                         !If Action = 2 (ChangeRec)
            DELETE(Sample)                      !If Action = 3 (DeleteRec)
           END
           ErrorCheck                !A generic error checking procedure
           BREAK                     !  and break out of process loop
   . .                               !End loop and case
```

You can see that the update procedure code using this method is a bit smaller, and easier to follow logically. There are two new data declarations: SavCRC is declared to save the original CRC calculation, and PassArray is an array declared OVER the file's record buffer. The PassArray declaration simply provides a way to pass the CRC16 procedure the entire record as an array of BYTEs, it does not allocate any memory.

A valid question at this point would be, "Why is the Sav:SaveRecord declared in this code with a STATIC attribute?" There are four reasons:

- A save area is still needed to temporarily keep the user's changes while concurrency checking calculations are being performed.
- Variables declared locally in a PROCEDURE are assigned memory on the stack when the PROCEDURE is called.
- The save area for a large record buffer could easily take more memory than is available on the stack. Therefore, the save area should be in static memory.
- In a PROCEDURE, only data structures (WINDOW, REPORT, FILE, VIEW, and QUEUE) and variables with the STATIC attribute are assigned static memory.

Of course, if the save area were declared in the global data section (between the keywords PROGRAM and CODE), or a MEMBER module's data section (between the keywords MEMBER and PROCEDURE), it would not need to be declared with the STATIC attribute—it would automatically be assigned static memory.

HOLD and RELEASE

A tool to prevent other users from making changes to a record while it is being updated is the HOLD statement. HOLD tells the following GET, NEXT, or PREVIOUS statement to get the record and set a flag that tells any other user attempting to get that record that it is in use—a "record lock." The record remains held until it is: explicitly released with a RELEASE statement; implicitly released by a PUT, or DELETE, of that record; or, implicitly released by retrieving another record from the same file.

The Clarion language supports multiple file systems through its file driver technology. Each file system may implement record locking in a different manner. Therefore, the actual effect of HOLD is dependent upon the file driver, which takes whatever action is appropriate to the file system. In some file systems, a HOLD on a record allows other users to read the record, but not to write to it. In others, HOLD blocks other users from any access to the record. Some file systems release the HOLD automatically if the system crashes, others don't and leave it flagged as held. The specific action of HOLD is described in each file driver's documentation.

If you HOLD a record when it is retrieved, and RELEASE it when you write it back, you can eliminate the need for the type of concurrency checking previously described. Is this a good idea, though? Depending upon the actual implementation of HOLD in the file system being used, the answer may be either Yes or No.

If your file system blocks other users from all access to the record, or there is the possibility a "system crash" could leave it in a HOLD state, the answer is probably No. This does not mean that you should not use HOLD at all. It does mean that you should not use HOLD where a record would be held during user input (an indeterminate amount of time). More likely, you would use HOLD during the concurrency check described above. This is to make sure that nobody changes the record between the second GET (for concurrency checking) and the PUT that writes the user's changes to disk.

Using HOLD in this manner only changes the code in the CASE ACCEPTED() OF ?OKButton.

```
OF ?OKButton                          !Screen completion field
 IF Action = ChangeRec                !If changing an existing record
  Sav:SaveRecord = Sam:Record         !Save changes made
  HOLD(Sample,1)                      !Hold while checking for changes
  REGET(Sample,SavRecPos)             !Get record from FILE again
  IF ERRORCODE()
   IF ERROR() = 'RECORD ALREADY HELD' !Has someone else got it?
    BEEP                             ! Alert the user
    SHOW(25,1,'Held by another station')
    SELECT(1)                        !   and start over
    CYCLE                            !    at first field
   ELSIF ERROR() = 'RECORD NOT FOUND' !Did someone else delete it?
    Action = AddRec                  ! change Action to add it back
    Sam:Record = Sav:SaveRecord      ! put them back in record buffer
   ELSE
    STOP(ERROR())                    !Stop on any other error
   END
  ELSIF SavCRC <> CRC16(PassArray)   !Compare CRC values for changes
   RELEASE(Sample)                   !Release the hold
   SavCRC = CRC16(PassArray)         ! Save new CRC value
   DISPLAY                           ! Display other's changes
   BEEP                              ! Alert the user
   IF MESSAGE('Changed by another station').
    SELECT(1)                        !   and start over
    CYCLE                            !    at first field
   ELSE                              !If nobody changed it
    Sam:Record = Sav:SaveRecord      ! put these changes back in buffer
. .
 EXECUTE Action                       !Execute disk write
  ADD(Sample)                         !If Action = 1 (AddRec)
  PUT(Sample)                         !If Action = 2 (ChangeRec)
  DELETE(Sample)                      !If Action = 3 (DeleteRec)
 END
 ErrorCheck                           !A generic error checking procedure
 BREAK                                !  and break out of process loop
```

This change puts a HOLD on the record only long enough to determine if it is the same record the user started with and write the changes to disk. If someone else has a HOLD on the record, the user is alerted to that fact and allowed to try again. If the record continually comes up as held by another station, then it has probably been left in a HOLD state by a system crash. In that case, the hold should be released by whatever action is appropriate for that file system. Code could be written to handle this eventuality, but it would be specific to the file system and this is “generic” example code.

If the prevention of record update conflicts is a “mission-critical” concern, then HOLD could be used to keep control of the record during user data entry. One trade-off with this use of HOLD is the nuisance of dealing with records that are left locked when users’ systems crash while records are held. Correcting that situation could involve some manual work with file system utilities, or could simply be a matter of specific coding considerations for the file system being used. Another concern with using HOLD this way comes when the file system being used does not allow other users to read the held records. The held records would seem to “disappear” then “reappear” from time to time as users HOLD records. Either way, this method should probably not be used unless the application really requires it.

To utilize this technique, the HOLD would have to be in the procedure which actually retrieves the record from the file. In most cases, that procedure would display some kind of scrolling list of records, usually displayed in a LIST box. The following example code demonstrates this.

```
OF ?List                                !LIST
 CASE EVENT()
  OF EVENT:Accepted                    !An existing record was selected
   GET(TableQue,CHOICE())              !Get record number from the QUEUE
   HOLD(Sample,1)                      !Arm the HOLD
   REGET(Sample,Que:RecPosition)       !  and get the record from the file
   IF ERROR() = 'RECORD ALREADY HELD' !Has someone else got it?
    BEEP                               ! Alert the user
    IF MESSAGE('Held by another station').
     SELECT(?List)                     !   to try again
     CYCLE
    ELSE                               !If no one else has it
     Action = ChangeRec                !Set up disk action for change
     Update                            ! and call the update procedure
    END
     !Code to handle other keycodes goes here
  END
```

This technique grossly simplifies the update procedure code, as in this example:

```
Update   PROCEDURE    !An update procedure
Screen   WINDOW
            !data entry controls go here
         END
   CODE
   OPEN(Screen)
   DISPLAY                            !Display the record on screen
   ACCEPT                             !Screen field process loop
    CASE FIELD()
    !Individual screen field edit code goes here
     OF ?OKButton                     !Screen completion field
      CASE EVENT()
       OF EVENT:Accepted
        EXECUTE Action                !Execute disk write
         ADD(Sample)                  !If Action = 1 (AddRec)
         PUT(Sample)                  !If Action = 2 (ChangeRec)
         DELETE(Sample)               !If Action = 3 (DeleteRec)
        END
      ErrorCheck                      !A generic error checking procedure
      BREAK                           !  and break out of process loop
   . .  .                             !End loop and case
```

The HOLD statement only allows each user to HOLD one record in each file. If you need to update multiple records in one file and you must be sure that no other user makes changes to those records while they are being updated, then you must LOCK the file.

LOCK and UNLOCK

The LOCK statement prevents other users from accessing any records in a file, until you UNLOCK it. Just like HOLD, the effect of LOCK is dependent upon the file driver which takes whatever action is appropriate to the file system. In some file systems, a system crash automatically unlocks the file, and in others it is left locked. The specific action LOCK takes is described in each file driver's documentation.

Because other users are completely barred from accessing records in the LOCKed file, LOCK is not commonly used. The most common use of LOCK would be to BUILD an INDEX prior to using it (and that is not even necessary if it is a "dynamic" INDEX). The type of "batch update processing" that would require a file to be LOCKed for a significant period of time is generally best left until after hours, when all users are gone. Other than to BUILD an INDEX, a file LOCK is usually only needed during Transaction Processing, which is the subject of a separate essay.

If an application truly demands a file LOCK, then the period of time during which the other users are denied access should be kept to an absolute minimum. The code between the file LOCK and its subsequent UNLOCK statement should not require any user input. This means that the code should be written such that an end-user cannot go to lunch leaving a file LOCKed. Specifically,

LOCK should come immediately before the BUILD occurs, and the file should be UNLOCKed as soon as it is complete.

```
ReportProc     PROCEDURE
Sample     FILE,DRIVER('TopSpeed'),PRE(Sam)  !A data file declaration
Field1Key   KEY(Sam:Field1)
Field2Ndx   INDEX(Sam:Field2)   !An INDEX
Record       RECORD
Field1        LONG
Field2        STRING(10)
           . .
Report     REPORT
              !Report declaration statements go here
           END
   CODE
   OPEN(Sample,42h)                          !Open Read/Write Deny None
   LOCK(Sample,1)                            !Lock the file
   IF ERROR() = 'FILE IS ALREADY LOCKED'     !Check for other locks
    BEEP                                     !Alert the user
    MESSAGE('Locked by another station')
    RETURN                                   !  and get out
   END
   BUILD(Sam:Field2Ndx)                      !Build the index
   UNLOCK(Sample)                            !Unlock the file
   OPEN(Report)
   SET(Sam:Field2Ndx)                        !Use the index
   LOOP
    NEXT(Sample)
    IF ERRORCODE() THEN BREAK.
    !Report processing code goes here
   END
```

This code opens the file in access mode 42h (Read/Write Deny None) for fully shared access. The LOCK is attempted for one second. If it is successful, the BUILD immediately executes. If the LOCK was unsuccessful, the user is alerted and returned to the procedure that called the report. Once the BUILD is complete, UNLOCK once again allows other users access to the file, and the report is run based on the sort order of the INDEX.

"Deadly Embrace"

There are two forms of "deadly embrace." The first occurs when two users attempt to LOCK the same set of files in separate orders of sequence. The scenario is:

User A locks file A

User B locks file B at the same time

User A attempts to LOCK file B and cannot because User B has it LOCKed

User B attempts to LOCK file A and cannot because User A has it LOCKed

This leaves both users "hung up" attempting to gain control of the files. The solution to this dilemma is the adoption of a simple coding convention: Always LOCK files in the same order (alphabetical works just fine) and trap for other users' LOCKs. This example demonstrates the principle:

```
LOOP
 LOCK(FileA,1)                      !Attempt LOCK for 1 second
 IF ERROR() = 'FILE IS ALREADY LOCKED'
  CYCLE                             !  and try again
 END
 LOCK(FileB,1)                      !Attempt LOCK for 1 second
 IF ERROR() = 'FILE IS ALREADY LOCKED'
  UNLOCK(FileA)                     !Unlock the locked file
  CYCLE                             !  and try again
 END
 BREAK                              !Break from loop when both locked
END
```

This code will eventually LOCK both files. If FileA is already locked by another user, the loop will try again. The one second pause allows the other user a chance to complete their action. If the first LOCK is successful, the LOCK on FileB is attempted. If FileB is already locked by another user, FileA is immediately unlocked for other user's use, then re-try sequence occurs again. The BREAK from the LOOP in this example is only allowed after both files are successfully LOCKed.

Mixing the use of HOLD and LOCK can result in the second form of "deadly embrace." In some file systems, LOCK and HOLD are completely independent, therefore it is possible for one user to HOLD a record in a file, and another user to LOCK that same file. The user with the HOLD cannot write the record back, or even RELEASE it, and the user with the LOCK cannot write to that held record.

This situation may be resolved in one of two ways:

- You may choose to never mix HOLD and LOCK on the same file. This limits you to the use of HOLD only (the most common solution), or LOCK only. This solution must be used in all applications that write to a common set of files.
- You may choose to always trap for held records while the file is LOCKed. This implies that you know how you want to deal with the "deadly embrace" record when it is detected.

The first solution is by far the more commonly used. The second takes you into an area of programming that is probably better served by Transaction Processing, which would include held record trapping.

One final note: Some file drivers can perform exclusive locks internally on update operations and STREAM. The use of a STREAM statement is not safe from deadlock on multiple files for write operations. You should use LOGOUT instead of LOCK in these situations.

Summary

- A File must be opened before its records may be accessed.
- The access mode determines the type of access DOS grants to the user opening the file and any other users.
- Multi-user programming must always take into consideration the possibility of multiple users accessing the same record at the same time.
- Concurrency checking is done to ensure that user updates don't overwrite other user's changes to the records.
- HOLD is most commonly used in conjunction with concurrency checking to ensure that no other user can change the record while it is being compared for previous changes.
- LOCK is most commonly used to gain exclusive control of a file while you BUILD an INDEX.
- The "deadly embrace" is a programming consideration most easily dealt with through the adoption of consistent program coding conventions.

7 - Developing Client/Server Applications

Introduction to Client/Server

Client/Server Defined

What exactly constitutes Client/Server computing? To some, it indicates the use of Groupware (programs such as Lotus Notes), or Object Linking and Embedding (OLE) between OLE client and OLE server applications, or just any program that splits task completion between client workstations and network servers. There are also many “buzzwords” that go along with the subject, such as “downsizing” and “rightsizing.”

At its most basic level, the broadest definition of Client/Server computing comes down to this: a Client workstation computer sends a message (requesting some kind of service) across a network of some kind to a Server computer, which processes the service request and responds with either a return message (returning requested data) or an acknowledgement. No matter what form it takes, the ultimate purpose of Client/Server computing is to make the most efficient use of the computing resources available, on an enterprise-wide basis.

Clarion is optimized to create business database applications. This means the most common type of Client/Server applications created with Clarion are the type where the program executes on a Client workstation and accesses a database server engine (frequently SQL-based) located on a database server in the network.

Therefore, for the purposes of this discussion, the definition of Client/Server computing that we will use is this: creating database applications that logically partition the workload between the Client workstations and the Server-based database engine so that maximum performance efficiency is maintained throughout the network.

Types of Client/Server Database Applications

There are two major categories of Client/Server database applications: On Line Transaction Processing (OLTP), and Decision Support Systems (DSS).

On Line Transaction Processing (OLTP)

OLTP applications are built to continually maintain and update the database in real time. This means OLTP applications require high performance from both the hardware and software so that users are not kept waiting for any requested information for more than a very few seconds.

Decision Support Systems (DSS)

DSS applications are usually used by a company’s management for stored or ad hoc queries to monitor the state of the company. This usually means the data does not need to be quite so “up to the minute” and response times are not nearly as critical. Frequently, DSS applications are built to access on-line archival data or a replication of the database that is updated only at timed intervals.

Clarion Client/Server Applications

Typically, Clarion-created Client/Server database applications fall into the On Line Transaction Processing (OLTP) category, rather than the Decision Support Systems (DSS) used for ad hoc queries. Your users can use Clarion's ReportWriter for Windows product to generate standard or ad hoc reports from the database (thus providing them DSS functionality).

SQL Database Engines

Most Client/Server applications are "front-ends" running on a Client workstation and requesting data services from a database engine (such as Oracle or Sybase) running on a network Server. Most of these database engines are accessed through Structured Query Language (SQL).

SQL was originally developed as an ad hoc query language to allow users to glean information from a database. Over the years, the SQL language has been enhanced to allow programmatic access to databases in addition to its original ad hoc query mission.

SQL has become the standard tool used to access the database engines used in Client/Server computing—programmatic SQL is the tool most frequently used to perform the data access chores in the programs that execute on Client workstations. However, SQL is not a full-featured programming language (for instance, it has no user interface design capabilities), so the SQL must be "embedded" into a more general purpose programming language (such as C++ or Clarion) to create a complete program.

Clarion and SQL

Clarion's file driver technology allows you to write applications that access SQL databases without the necessity of writing any SQL at all—the Clarion file driver for any SQL database communicates with the back-end database server by automatically generating the SQL statements that the database engine requires from your standard Clarion language file I/O syntax. Since the Clarion SQL file drivers are specifically designed to "talk to" a single SQL back-end database (except the ODBC driver, of course, which is a generic SQL file driver), the SQL it generates is optimized to use features specific to that back-end database. This means that a Clarion programmer is not required to learn (or use) SQL at all to write efficient Client/Server applications.

The fact that a Clarion programmer is not required to write SQL does not in any way limit the capability of Clarion to generate Client/Server programs. You can embed your own SQL statement to extend what the file driver does for you, or you can take full control of the database yourself to accomplish any task that needs doing.

There are several ways to directly embed your own SQL statements into a Clarion program, if you so choose. This gives you, the Clarion programmer, the flexibility to just "let it happen" in a standard fashion (by letting Clarion's file driver handle all the SQL), or to "make it happen" any other way you choose by directly writing your own SQL, as appropriate to the requirements of your individual application.

PROP:SQL is a property of a FILE or VIEW structure that allows you to send any SQL statement directly to the back-end database server. PROP:SQLFilter is a property of a VIEW structure that allows you to append your own SQL WHERE clause to the generated SQL sent to the back-end database. Beyond these, there are also several other properties available that are used to enhance the SQL generated by the file driver (some are file-driver-specific). In either case, the file driver places the returned data set from the back-end database in the appropriate FILE record buffer, allowing you to use standard Clarion syntax to manipulate its value, or assign the data values to other fields.

Database Design and Network Traffic

The most important aspect to creating efficient Client/Server applications is the design of the database. When properly designed, the database more readily allows you to write your applications such that network traffic can be minimized. When network traffic is kept to a minimum, data response times (the most critical aspect of OLTP systems) can be kept as fast as possible, even when the system is scaled up to the enterprise level.

Referential Integrity Handling

Maintaining the Referential Integrity (RI) of a database is a key element to Relational Database design. Referential Integrity means that, for every One-to-Many (Parent-Child) relationship between tables in the database there exists a Parent record for every Child record (no “orphan” records). To put it in more formal terms, there must be a valid Primary Key value for every existing Foreign Key in the database.

“Orphan” records can occur when the Parent record is deleted, or the Primary Key value (which provides the link to the Foreign Key in the Child record) is changed. Preventing these “orphan” records requires that the database contain rules stating what action will occur when the end-user attempts to delete the Parent record, or change the Primary Key value.

The most common RI rules are “restrict” (do not allow the delete or change) and “cascade” (delete the related Child records or change their Foreign Key values to match the new Primary Key value in the Parent record). A rarely used rule is “clear” (change the Foreign Key values to NULL when the Parent record is deleted or the Primary Key value in the Parent record changes).

RI constraint enforcement is best handled in Client/Server applications by specifying the RI rules on the back-end database server, usually by defining Triggers, Stored Procedures, or Declarative RI statements. By doing this, the database server can automatically handle RI enforcement without sending any Child records across the network to the Client application for processing. For example, if the rule for Delete is “cascade,” the database server can simply perform all the required Child record deletions when deleting the Parent record from the database—without sending anything back across the network to the Client application.

In the Clarion Dictionary Editor, when you establish a relationship between two files (tables), you can also specify the RI rules for that relationship. Since the database server will actually be handling the RI functionality, the most appropriate way to specify the RI rules in the Clarion Data Dictionary would be to specify “no action” so the Client does nothing.

Data Validation

Data validation means the enforcement of business rules (that you specify) as to what values are valid for any particular field in the database. Typical data validation rules enforce such things as: a field may not be left empty (blank or zero), or the field's value must be either one or zero (true or false) or within a certain specified numeric range, or the field's value must exist as a Primary Key value in another file (table).

These data validation rules can be specified either in the Client application or in the back-end database server. The best way to handle implementing data validation rules in your Client/Server applications, so as to generate minimal network traffic, is to specify the business rules in both the Client application and the database server:

- By enforcing data validation rules in the Client application you ensure that all data sent to the back-end is already valid. By always receiving valid data the database server will not generate error messages back to the Client application. The net effect of this is to reduce the network traffic back from the database server.
- By enforcing data validation rules on the back-end database server you ensure that the data is always valid, no matter what application is used to update the database—even updating the data with interactive SQL cannot corrupt the data. Therefore, you are covered from both directions.

Enforcing these rules in both your Clarion applications and the database server may seem like a lot of work. However, the Clarion Data Dictionary Editor allows you to specify the most common rules by simply selecting a radio button on the Validity Checks tab of the affected field's definition. By doing this, the actual code to perform the data validation is written for you by the Application Generator's Templates.

Clarion Language Client/Server Support

The Clarion language contains a number of statements that are explicitly designed to support Client/Server application programming.

The VIEW Structure

The VIEW structure is a data structure that automatically defines two standard relational database operations: the PROJECT and JOIN operations. A VIEW will also automatically FILTER and ORDER the result set. The back-end database server performs all these operations, returning only the result set that the Client application needs to perform its work.

Although it is actually possible to create a VIEW structure that JOINs tables from different back-end database servers (such as Oracle and AS/400), this would not be a very efficient way to write a Client/Server application because the JOIN (and any filtering) would have to be processed on the Client, eliminating the advantage of Client/Server computing. Therefore, we will not address this possibility in this article.

PROJECT

A relational PROJECT operation tells the back-end database server to only return to the Client application a specific sub-set of the columns in a table (thereby reducing network traffic). For example, the following VIEW structure will return only the specified fields (columns) from the Students file (table):

```
MyView  VIEW(Students)
          PROJECT(STU:LastName,STU:FirstName,STU:Major)
        END
JOIN
```

The relational JOIN operation automatically joins together related rows from multiple tables into a single result set which the database server returns to the Client application. The VIEW structure defaults to a left outer join unless the INNER attribute is specified on the JOIN structure. A left outer join returns all of the outer rows, whether there are related inner rows to return or not. For those outer records without related inner records, the fields in the inner record are left empty (blank or zero). For example, the following VIEW structure will return all records (rows) in the Students file (table), whether or not there are related Majors file records:

```
MyView  VIEW(Students)
          PROJECT(STU:LastName,STU:FirstName,STU:Major)
          JOIN(MAJ:KeyNumber,STU:Major)
            PROJECT(MAJ:Description,MAJ:Number)
          END
        END
```

Adding the INNER attribute to the JOIN structure specifies an inner join which returns only those outer rows with related inner rows. For example, the following VIEW structure will return only the records (rows) in the Students file (table), where there are related Majors file records:

```
MyView  VIEW(Students)
          PROJECT(STU:LastName,STU:FirstName,STU:Major)
          JOIN(MAJ:KeyNumber,STU:Major),INNER
            PROJECT(MAJ:Description,MAJ:Number)
          END
        END
```

FILTER

The FILTER attribute on a VIEW allows you to specify a conditional expression to filter out unwanted records. This will generate a WHERE clause in the generated SQL SELECT statement. For example, the following VIEW will return only those Students file records whose last name is “Taylor”:

```
MyView  VIEW(Students),FILTER('STU:LastName = ''Taylor''')
          PROJECT(STU:LastName,STU:FirstName,STU:Major)
        END
```

ORDER

The ORDER attribute on a VIEW allows you to specify the sort order of the result set returned by the database server. This will generate an ORDER BY clause in the generated SQL SELECT statement. For example, the following VIEW returns the Students file records sorted in descending last name and ascending first name order:

```
MyView  VIEW(Students),ORDER('-STU:LastName,+STU:FirstName')
          PROJECT(STU:LastName,STU:FirstName,STU:Major)
        END
```

Naturally, all these attributes can be combined so that the result set returns to the Client application filtered, ordered, projected and joined. By allowing the back-end database server to do this work, the only network traffic generated is the minimum necessary to give the Client application the requested data.

The BUFFER Statement

The Clarion BUFFER statement can have a tremendous impact on Client/Server application performance. BUFFER tells the file driver to set up a buffer to hold previously read records and a read-ahead buffer for anticipated record fetches. It also specifies a time period during which the buffered data is considered to be valid (after which the data is re-read from the back-end database server).

When the file driver knows it has buffers to hold multiple records it can optimize the SQL statements it generates to the back-end database server. This allows the back-end database server to return a set of records instead of a single record at a time (also called “fat fetches”). The net effect of this is to change the pattern of network traffic from many small pieces of data to fewer but larger chunks of data, making for more efficient overall network utilization. The most common use of BUFFER would probably be in procedures which allow the end-user to browse through the database.

By setting up buffers to hold already read records, the Client machine fetches records from the local buffer when the user has paged ahead then returns to a previous page of records, instead of generating another request to the back-end database server for the same page of records. This eliminates the network traffic that would normally be generated for subsequent requests for the same set of records.

Setting up read-ahead buffers enables the Client application to anticipate the user’s request for the next page of records and receive them from the back-end database server while the user is still examining the first page. Therefore, when the user finally does request the next page, those records are also fetched from the local buffer on the Client machine, giving the end-user apparently instantaneous database retrieval.

For example, the following BUFFER statement tells the file driver to consider 10 records as a single “page,” to buffer 5 previously read pages, to read 2 pages ahead, and to consider the buffered records valid for 5 minutes (300 seconds):

```
BUFFER(MyView,10,5,2,300) !10 recs per page, 5 pages behind, 2 read-ahead
                          !with a 5 minute timeout
```

Embedded SQL in Clarion

Clarion's file driver technology lets the Clarion programmer learn and use just the Clarion language's file Input/Output syntax, no matter which file system contains the data. The file driver automatically converts the Clarion language statements into whatever form the file system requires for its fulfillment of data requests. Therefore, for file drivers which interface to SQL-based database servers, the file driver itself generates the necessary SQL statements to retrieve the data from the database server.

Given all the information that the file driver can obtain from FILE structure declarations, VIEW structure declarations, and the BUFFER statement, the resulting SQL generated by the file driver can be quite efficient. However, there are still times when an experienced and knowledgeable SQL programmer will want to extend the functionality of the file driver, or take full control to accomplish a task that is best accomplished in SQL—some task that the file driver would not normally generate (such as deleting an entire block of records at once, or updating a single field in all rows of a table to some new value). To cover these circumstances, the Clarion language provides a mechanism to allow sending any SQL statement directly to the back-end database server: PROP:SQL.

PROP:SQL

The syntax of PROP:SQL is the same as any other Clarion language property assignment statement, with the target being the label of any FILE (or VIEW) declaration that uses the SQL file driver. For example:

```
MyFile{PROP:SQL} = 'SELECT * FROM SOMETABLE'
```

This statement sends the specified SQL SELECT statement to the back-end database server. Note that the target of PROP:SQL is MyFile while the SQL SELECT statement is directed to SOMETABLE. You can send any SQL statement to PROP:SQL, regardless of whether the target of PROP:SQL is affected by the SQL or not—the target file is just the mechanism whereby the correct file driver processes your SQL. Since this SQL SELECT statement will return a result set, you must use the Clarion NEXT statement to retrieve the result set one record at a time, and the FILE declaration referenced in the NEXT statement must have the same number of fields as SOMETABLE. Obviously, this specific example does not demonstrate the preferred method of working in Clarion, since the file driver itself will more than adequately generate such simple SQL SELECT statements for you.

The real usefulness of PROP:SQL is to perform SQL functions that have no direct corollary in the Clarion language. One prime example of this is table (file) creation on the database server. The Clarion language CREATE statement will certainly allow you to create new tables in the database and the file driver will generate an appropriate SQL CREATE statement to perform the task. However, the Clarion CREATE statement is limited to the information stored in your FILE declaration. It is much better to use PROP:SQL to send an SQL CREATE statement to the database server, because the SQL CREATE can then implement any data validation constraints or triggers that the database server supports, allowing you to make full use of the capabilities of the database server.

For example, the following code creates a Students table with First and Last Names, Major, and Dormitory assignment columns:

```
MyView{PROP:SQL} =    'CREATE TABLE  STUDENTS'                 &  |
                       '(IDNUMBER    INTEGER NOT NULL, '       &  |
                        'LASTNAME    VARCHAR(25) NOT NULL, '   &  |
                       'FIRSTNAME    VARCHAR(25) NOT NULL, '   &  |
                           'MAJOR    VARCHAR(10) NOT NULL, '   &  |
                       'DORMITORY    INTEGER, '                &  |
                     'PRIMARY KEY    (IDNUMBER), '             &  |
                     'FOREIGN KEY    MAJORS_IN (MAJOR), '      &  |
                      'REFERENCES    MAJORS, '                 &  |
                       'ON DELETE    RESTRICT)'
```

This SQL CREATE statement specifies two items that are not possible to specify using the Clarion CREATE statement. The NOT NULL attribute specifies that data must be present in these columns whenever a new record is added to the database. The ON DELETE clause specifies that the RESTRICT Referential Integrity constraint applies when deleting a Majors record, so that no Student record can have a Major field value that doesn't exist in the Majors table.

Another primary use of PROP:SQL is to call any stored procedures that you have created in your back-end database. In most SQL-based database servers, a stored procedure is a pre-compiled set of SQL statements that allow you to pre-define actions for the database engine to take when the procedure is called. Since stored procedures are a part of the database, they can be used by any application that accesses the database, which helps enforce consistent interaction with the database across all applications.

PROP:SQLFilter

PROP:SQLFilter is very similar to PROP:SQL. PROP:SQLFilter allows you to specify a filter condition for a VIEW structure using SQL syntax instead of the Clarion syntax required in the FILTER attribute.

By default, the PROP:SQLFilter filter condition overrides any expression in the FILTER attribute. However, by beginning your PROP:SQLFilter expression with a plus sign (+), the file driver will append your PROP:SQLFilter statement to the FILTER attribute's generated SQL.

The advantage of using PROP:SQLFilter instead of the FILTER attribute is to allow the WHERE clause to use the database-specific or SQL-specific capabilities that you can include by using your own SQL (such as an SQL IN clause). For example, the following code uses a FILTER condition to limit the VIEW to only those students with "Computer Science" as their major:

```
MyView  VIEW(Students),FILTER('STU:Major = ''Computer Science''')
          PROJECT(STU:LastName,STU:FirstName,STU:Major)
        END
```

The following code replaces the FILTER with PROP:SQLFilter:

```
MyView  VIEW(Students)
          PROJECT(STU:LastName,STU:FirstName,STU:Major)
        END
  CODE
  OPEN(MyView)
  MyView{PROP:SQLFilter} = 'Students.Major = ''Computer Science'''
```

Both of these examples return the same result set to your Clarion application.

The following code uses the FILTER condition to limit the VIEW to only those students with "Computer Science" as their major, then appends an SQL IN clause to the generated WHERE clause to further limit the result set to only those Computer Science students who also live in dormitories 1, 5, or 9:

```
MyView  VIEW(Students),FILTER('STU:Major = ''Computer Science''')
          PROJECT(STU:LastName,STU:FirstName,STU:Major,STU:Dormitory)
        END
  CODE
  OPEN(MyView)
  MyView{PROP:SQLFilter} = '+Students.Dormitory IN (1, 5, 9)'
```

NULL Data Handling

One concept common in SQL-based database servers is the concept of NULL data. The concept of a null "value" in a field of a FILE or VIEW indicates that the user has never entered data into the field. Null actually means "value not known" for the field. This is completely different from a blank or zero value, and makes it possible to detect the difference between a field which has never had data, and a field which has a (true) blank or zero value. The Clarion language supports NULL data handling through the NULL, SETNULL, and SETNONULL procedures.

In expressions, null does not equal blank or zero. Therefore, any expression which compares the value of a field from a FILE or VIEW with another value will always evaluate as unknown if the field is null. This is true even if the value of both elements in the expression are unknown (null) values. For example, the conditional expression Pre:Field1 = Pre:Field2 will evaluate as true only if both fields contain known values. If both fields are null, the result of the expression is also unknown.

```
Known = Known           !Evaluates as True or False
Known = Unknown         !Evaluates as unknown
Unknown = Unknown       !Evaluates as unknown
Unknown <> 10           !Evaluates as unknown
1 + Unknown             !Evaluates as unknown
```

The only four exceptions to this rule are boolean expressions using OR and AND where only one portion of the entire expression is unknown and the other portion of the expression meets the expression criteria:

```
Unknown OR True        !Evaluates as True
True OR Unknown        !Evaluates as True
Unknown AND False      !Evaluates as False
False AND Unknown      !Evaluates as False
```

Support for null "values" in a FILE or VIEW is entirely dependent upon the file driver. Most SQL-based database systems support the null field concept, while most non-SQL databases do not.

You use the Clarion NULL procedure to detect whether a data item returned from the back-end database is null or not. For any fields that should remain null when you re-write the data to the database, you must explicitly call SETNULL just before writing the data back to the database. For example:

```
NEXT(MyTable)                       !Get data
MyFieldFlag = NULL(MyField)         !Remember whether a field is null or not
  !
  !Process the data
  !
IF MyFieldFlag    AND NOT MyField   !Detect null field still empty
  SETNULL(MyField)                  ! and reset it to null
END
PUT(MyTable)                        !Write the data back
```

Error Handling

Whenever any I/O operation executes there is the possibility of an error condition occurring. No matter which back-end database server you use, the Clarion file driver for that database maps the most common errors to the appropriate standard Clarion error codes. However, there are always some errors for which there is no direct Clarion equivalent.

Whenever an error occurs for which there is no direct Clarion equivalent, the Clarion ERRORCODE procedure returns 90 and the ERROR procedure returns “File Driver Error.” To determine the exact error returned by the back-end database server in this case, Clarion provides the FILEERRORCODE and FILEERROR procedures.

The FILEERRORCODE and FILEERROR procedures return the back-end database server’s error values when the current Clarion ERRORCODE is 90. This allows you to detect any error the database server can issue, then look in the back-end database server’s documentation for the possible causes and remedies for whatever error has occurred.

Index:

”dynamic” INDEX....62
Base CLASS....20
BUFFER....87
Cascade....57
Clarion and SQL....83
CLASS....14, 33
CLASS Methods....33
CLASS Properties....33
Client/Server....82
Composition....22, 44
Concurrency Checking....70
Constructors....17, 36
Creating Objects....34
data declaration....6
Data Validation....85
Database Design....50
Deadly Embrace....80
Derived CLASS....20
destructors....36
Destructors....17
DISPOSE....18
DSS....82
Embedded SQL....88
Encapsulation....14, 31
Error Handling....91
Field Qualification Syntax....16
File Access Methods....62
File Relationships....52
FILEERROR....91
FILEERRORCODE....91
FILTER....86
Foreign Key....51
HOLD....75
INDEX....62
inheritance....39
Inheritance....20, 31
Instantiation....15
KEY....62
late binding....32
Late Binding....28
Local Derived Methods....29, 48
LOCK....78
MAP....7
MEMBER....7
MODULE....7
Multiple Inheritance....21
Multi-User Considerations....69
Network Traffic....84
NULL Data Handling....90
Nullify the Foreign Key....59
Object Oriented Programming....13
Objects....30
OLTP....82
OOP....13
Opening Files....69
ORDER....87
Overriding....41
PARENT....25
Polymorphism....25, 31, 46
PRIVATE....19, 37, 38
PROCEDURE....6
Procedure Overloading....25
program....6
PROJECT....86
PROP:SQL....84, 88
PROP:SQLFilter....89
PROTECTED....23, 37, 39
Public....37
Random File Access....66
reference variables....35
Referential Integrity....55, 84
Relational Database Design....50
RELEASE....75
REPLACE....43
Restrict....56
SELF....17
Sequential File Access....63
UNLOCK....78
Variables....6
VIEW....85
Virtual Methods....26, 46