INTEGRATING AN OBJECT-ORIENTED PROGRAMMING LANGUAGE SYSTEM WITH A DATABASE SYSTEM

By

HUI-CHEN NEE

Bachelor of Engineering

Tamkung University

Taiwan, R.O.C.

1987

Submitted to the faculty of the
Graduate College of the
Oklahoma State University
In partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1991

INTEGRATING AN OBJECT-ORIENTED

PROGRAMMING LANGUAGE SYSTEM

WITH A DATABASE SYSTEM

Thesis Approved:

[illegible]

Thesis Advisor

Huizhu Lu

J P Chandler

[illegible]

Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my appreciation to the chairman of my advisory committee, Dr. George Hedrick, for his patience, guidance, counsel, and understanding provided throughout the course of my graduate study. I also like to extend my thanks to Dr. Huizhu Lu and Dr. John Chandler for their advice and close reading of the thesis, and for serving as members of my graduate committee.

I also like to express my thanks to my parents, Chinghui Nee, and Shunu Chou for their love and support during my thesis writing.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Motivation

Object-oriented programming is an important paradigm for the software application challenges of the 1990s; object-oriented goals should improve the software development process resulting in the development of new and better applications. These applications include database management system (DBMS), artificial intelligence (AI), and computer-aid design (CAD). Object-oriented programming offers many advantages other than the traditional programming for the applications. The result is software that should be easier both to extend and to maintain as well as applications that are richer, easier to use, as well as more flexible (Wu, 1990).

Traditional relational database often both reduce the time required for application development and improve data sharing among applications. The relational data model has advantages and disadvantages. The primary advantage is that it is simple and powerful, which is why it is so popular among DBMS. However, the relational model is too simple, forcing programmers to work at too low a level of representation. Real-world applications are seldom so simple. Programmers need to translate complex models into the relational models. The primary disadvantage is that a relational database has only a limited set of data types (integer, string, and date), and it has a limited set of operations on these data types (retrieving the data, storing the data). Another disadvantage is the need to normalize data. Combining object-oriented programming concepts with database concepts can resolve these problems. Object-oriented programming can offer features similar to flexible abstract data type (ADT) facilities, using the ability to encapsulate data and operations by the message metaphor. The advantages of the combined object-oriented programming and database can offer not only the ability to represent and manage the complex relationships among data, but also the ability to describe almost any real-world entity as an object. The above benefits of object-oriented database should improve the current DBMSs.

Preliminary Literature Review

Object-Oriented Paradigm

The object-oriented paradigm applies to most major software fields, including languages, database management, artificial intelligence, computer-aided design, and manufacturing (CAD/CAM). The result is software that should be easier both to extend and to maintain as well as applications that are richer, easier to use, as well as more flexible (Wu, 1990).

The traditional record-oriented databases, e.g. hierarchical, network, and relational DBMS are limited in their abstractions and representational power. David Maier presents an object-oriented model offering more and better benefits than the current models. An object-oriented database can support the construction of objects and supertype and subtype hierarchies, which are useful for managing the complex data found in software environments. It can store not only complex application components but also large data structures (Maier, 1986). It also enables support of complex applications not supported well by the other models, enhanced programmability and performance, improve navigational access, and simplify concurrency control (Danforth, 1988) (Garza, 1988) (Hornick, 1987).

Object-Oriented Design

OODBPL introduces an object-oriented database programming (Suad, 1988). Two books provide techniques of object-oriented software construction; the design begins by defining objects and relationships. Next, life cycles of the objects are then describes in state models to define the events acting on the objects. The last step is process definition, based on the objects and their life cycles. Unlike the functional decomposition and event-response methods, the object-oriented approach results in minimal data-driven code that remains stable even when requirements are changing (Meyer, 1988) (Sally, 1988).

Integrated

Baroody examines object-oriented programming as an implementation technique for database systems. The object-oriented approach encapsulates the representations of entities and relationships with the procedures that manipulate them (Baroody, 1981). There are two ways to approach object-oriented DBMS : extending an object-oriented programming language or extending a relational DBMS. The first way is by using language: database functionality, such as persistence, authorization, and concurrency, is provided as needed for individual objects. An extended object-oriented programming language (OOPL) efficiently navigates individual objects and has no inherent limits on functionality. The method is good for sharing, querying and optimization but bad for transparency and flexibility (Kim, 1988). The second way extends the relational model with new types, operations, and access methods, which improves flexibility but can prevent method optimization (Andrews, 1987). Bernstein states that integrated database support will greatly reduce the efforts spent in maintaining this data and improve productivity for software product development (Bernstein, 1987). Agrawal develops a database system and environment based on the object paradigm. It begins with the object-oriented facilities of C++ and extends them with features to support the needs of databases (Agrawal, 1987). Kem shows how abstract data types integrated into the structurally object-oriented model can support engineering applications (Kem, 1987).

Technology and Prototype

Toby Bloom presents in an object-oriented database paradigm, a real-world entity is modeled as an instance (object) of a class which has a number of attributes (properties) and operations (methods) applicable to the objects (Bloom, 1987) (Penney, 1987) (Andrews, 1987) (Fishman, 1987) (Woelk, 1986). A class should inherit properties and methods from other classes. Banerjee introduces the fundamental data modeling concepts generalization and aggregation that are built into the object-oriented paradigm (Banerjee, 1987). Further, the inheritance mechanism makes it possible for applications to define new classes and have them inherit properties from existing classes, and this makes the applications easily reusable and extendible (Liu, 1988).

Comparison

Duhl and Rubenstein compare the object-oriented database and relational database. The results support strong evidence that object-oriented databases are better than relational database systems (Duhl, 1988) (Rubenstein 1987).

Problem Statement

Recent research in record-oriented database modeling has pointed to the lack of flexibility and expressiveness of these models. Object-oriented models, however, provide the database designer with expressive tools for conceptual modeling and provide abilities not available in the traditional models.

The following problems seek to use the relational database models as generic design data models:

1. The relational data models are all based on the notion of the logical record. But most of the world does not consist of records. The relational data models express the semantics of complex objects with difficulty because they possess only a table model for data storage.
2. Relational data models are limited in their abstractions and representational power. It is difficult to manage the complex and large software.
3. A relational data model is an inflexible model; it makes an expansive modification and difficult to reusable.

The object-oriented database should resolve the above problems.

1. Object-oriented programming can model real-world entities appropriate to the user's requirements. Object-oriented data models provide a rich set of relationships among real world entities, including the generalization, specialization, and aggregation relationships.

2. In an object-oriented data model, objects may exist in aggregation hierarchies which provide the capability to aggregate different types of multimedia information such as text, sound, and complex graphics drawings. It also supports dynamically varying data size, new data types and complex data relationships.
3. Object inheritance increases extendibility and reusability and saves the cost of maintenance.

The goal of this thesis is to combine an object-oriented programming system with a database system and improve the relational database. The object-oriented data model is written using an object-oriented programming language (Turbo C++). The data model is defined, manipulated, queried based on C++ and support data encapsulation and inheritance. After developing and designing the object-oriented data model, we apply this model to water resource data and compare this object-oriented DBMS with a relational DBMS. The object-oriented DBMS shows better performance than a relational DBMS as data type is complex.

Scope and Outline

The thesis is organized in the following manner: Chapter 1 includes an introduction and a literature review. Chapter 2 reviews the object-oriented programming and the development of databases. Chapter 3 compares structure programming vs object-oriented programming and relational database vs object-oriented database. Chapter 4 presents an object-oriented data model design and implementation. Chapter 5 is the conclusion, summary, and discusses future work to extend the work described in the thesis.

CHAPTER II

RELATED STUDIED

Object-oriented Programming

An object-oriented programming language must be object-based, provide classes, and support inheritance.
object-oriented = object-based + classes + inheritance.
The object-oriented language classification is as shown in Table I.

TABLE I

OBJECT-ORIENTED LANGUAGE CLASSIFICATION

	Object	Classes	Inheritance
Traditional languages (C, Pascal)	No	No	No
Object-based (Ada)	Yes	No	No
Class-based (CLU)	Yes	Yes	No
Object-oriented languages (C++,Smalltalk)	Yes	Yes	Yes

Basic object-oriented programming concepts include encapsulation, abstraction, modularity, and inheritance (Danforth, 1988).

Encapsulation is the process of hiding all of the details of an object that do not contribute to its essential characteristics; typically, the structure of an object is hidden, as well as the implementation of its methods.

Data abstraction could be considered a way of using information hiding. A programmer defines an abstract data type (ADT) consisting of a set of properties and a set of methods used to access and manipulate the data.

Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.

Inheritance is a mechanism for sharing properties and methods among classes, subclass, and objects automatically. A specialization of an existing class is called a subclass. The subclass inherits properties and methods from its superclass. The subclass may add properties and methods that are appropriate to more specialized objects.

Object-oriented programming languages have many advantages over than traditional structure programming languages. Encapsulation and data abstraction increase reliability and help decouple procedural and representational specification from implementation. Inheritance increases flexibility by allowing the addition of new classes of data types without having to modify the existing code. Inheritance also permits a code to be reused.

The basic elements of object-oriented programming language are objects and methods, classes and class hierarchy, and inheritance (Goldberg, 1983) (Cargill, 1986) (Meyer, 1987).

Objects and Methods

In object-oriented systems, all conceptual entities are modeled as objects. We define a record as a set of fields which corresponds to a tuple in a relational data model and an object in an object-oriented data model. A simple object could be an integer, a boolean value, a real value, or a string. A complex object may exist in aggregation and hierarchies which provides the capability to associate different types of multimedia information such as text, video, and complex graphics drawings. Object behavior is described by a combination of properties and methods. Properties represent static behavior and methods represent dynamic behavior.

The behavior of an object is encapsulated in methods. Methods consist of codes that manipulate or return the state of an object. Methods are a part of the definition of the object. However, methods, as well as objects, are not visible from outside the object (Kim, 1989).

<u>Classes Hierarchy and Inheritance</u>

Grouping objects into classes helps avoid the specification and storage of much redundant information. We define a record type as a group of records with the same data types. It is a relation in a relational data model and a class in an object-oriented data model. A class has two components associated with it: a list of properties and a set of methods. Properties of a class are described by the instance variables defined on the class. Methods are the operations that are performed on the instance variables of the class. The domain of methods could be multiple classes. The range of a method is a class. A method can be considered as a functional object. A class could be a system-defined class, or a user-defined class. A system-defined class, such as a class of integers or strings. A user-defined class such as a class of theses or students. The objects of a class are called its instances. Associated with each class is set of instance variables that describe the state of the instances of a class (Shriver, 1988).

The distinction between classes and objects is that objects are run-time elements that will be created during a system's execution; classes are a purely static description of a set of possible objects.

The concept of class hierarchy is based on two fundamental class types: (1) the IS-A hierarchy which describes generalization and specialization relationships. (2) the IS-PART-OF hierarchy which describes aggregation relationship. The IS-A hierarchy where a class has subclass associated with it. The subclass inherits all of the properties and methods associated with its superclass. For example, the class of all people has the class of all students as its subclass. This subclass will inherit all the properties from its superclass plus have additional properties such as ID# and major. The IS-A hierarchy described is in Figure 1.

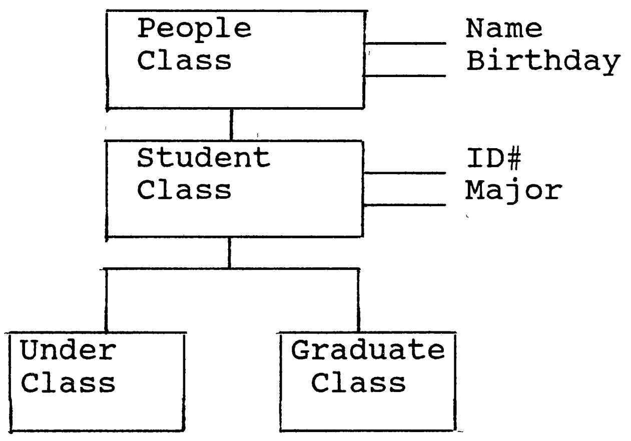


Figure 1. IS-A Hierarchy

The second class hierarchy is the IS-PART-OF hierarchy. An object of a class is considered to be the aggregation of a set of objects, each of which belongs to same classes. Such an aggregate object also is called a composite object. For example, a thesis object, which is an object of the thesis class, consists of a title, table of contents, a set of chapters, and a set of references. A chapter object that belongs to the chapter class has a title and a set of sections as its components. A section object that belongs to the section class has a title and a set of paragraphs as its components. The IS-PART-OF hierarchy is in Figure 2.

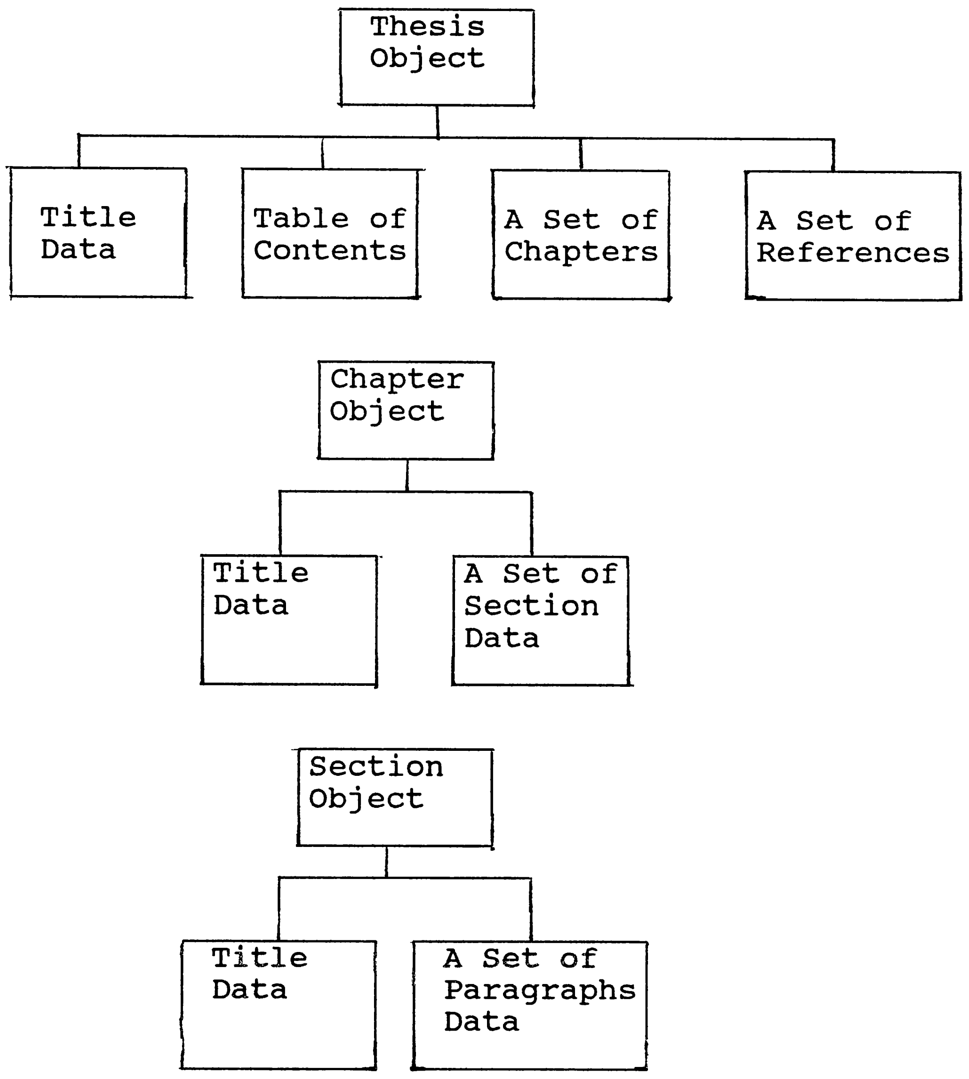


Figure 2. IS-PART-OF Hierarchy

Inheritance allows objects to be arranged in taxonomies in which the more specialized objects inherit properties, i.e., the properties and the methods of more generalized objects. Similar objects with a few different properties can be modeled by specialized classes from this superclass. Derived classes can be used to construct heterogeneous data structures such as lists with different types of elements because a pointer to a class can point to any object whose type is derived from this class.

The Development of Database System

The major goals of database are both reduce the time and costs required for application development and improve data sharing among application. Hierarchical data models appeared in the late 1960s and consist of an ordered set of trees, the mean is an ordered set consisting of multiple occurrences of a single type of tree. Relational data models appeared in the late 1970s and provided many benefits over the hierarchical data model. The table and the relational form are simple and powerful. The traditional relational databases are designed for business-like applications. As is well known, the relational data models express the semantics of complex objects with difficulty because they process only a table model for data storage. Object-oriented databases appeared in the late 1980s. The applications of the 1990s require support for complex data structures, large data access, and better performance. The change from relational data models to object-oriented data models is the same as the change from hierarchical data models to relational data models.

The Relational Data Model

The relational database approach created by E.F Codd and developing since 1970 is a considerably different approach to the logical description and manipulation of data. It attempts to avoid many of the disadvantages mentioned and to provide advantages in data independence, ease to use and user friendly, database processing power, and security controls. It views the logical database as a simple collection of two-dimensional tables called relations. These tables are flat in that no repeating groups are involved. They are easy understood and handled by users with little training in programming, and they involve no consideration of positional, pointer, or access path aspects (Date, 1990).

A relation or table is a two-dimensional array with the following characteristics: (Date, 1990)

1. Each entry in the table is one data item; there are no repeating groups. Each domain Di must be a simple domain; that is, it does not represent another relation. A relation is said to be normalized if it has no repeating groups; otherwise it is said to be unnormalized.

2. Each column, the domain, is assigned a distinct name, a domain name, and is made up of values of the same data item.
3. All rows or tuples are distinct; duplicate are not permitted.
4. The rows and the columns can be ordered in any sequence at any time without affecting the information content or the semantics involved.

Codd suggests two languages to carry out these operations in logical terms: the relational algebra and the relational calculus. The basic relational algebra operators involved union, intersection, difference, extended cartesian product, selection, projection, join, and division. A relational algebra operator takes one or more relations as its operands and produces a relation. A relational calculus is a mathematically-oriented notation for defining a relation to be derived from existing relations in the data model. It permits users to describe what they want to obtain without indicating many details of how to obtain it.

A relational data model is viewed by users as a collection of normalized relations of various degrees manipulated by powerful operators for extracting columns and joining them.

The Object-Oriented Data Model

The object-oriented database is written using an object-oriented programming language depending on database systems for data storage and retrieval. Object-oriented data models add to traditional programming languages such concepts as persistence, sharing, transactions, and efficient access to large amounts of data. They also add to traditional database concepts such as abstraction, extensible typing, and procedures.

Object-oriented data models produce relational data models that close real-world applications and reduce the normalization problems often appeared in relational data model design. Object-oriented data models support code reuse, code maintenance, and modularity. An object-oriented data model, that is, a data model which is based on object-oriented concepts, provides the generalization and the aggregation relationships. There are very useful semantic relationships between objects (Maier, 1986) (Zdonik, 1990) (Kim, 1989).

An object-oriented database is based on a set of defined classes. Object behavior is described by a combination of properties and methods. Properties represent static behavior and methods represent dynamic behavior. The object-oriented data model is based on two fundamental class types : (1) the IS-A hierarchy which describes generalization and specialization relationships; (2) the IS-PART-OF hierarchy which describes aggregation relationship.

CHAPTER III

COMPARISON OF TWO SYSTEMS

Traditional vs OOP Concepts

From a traditional programmer's viewpoint, some object-oriented concepts names are replaced with different names than in the traditional concepts. In fact, some object-oriented concepts are similar to traditional programming approaches. We will compare the traditional and object-oriented terms and concepts (Bloom, 1987) (Duhl, 1988) (Date, 1990).

Class and properties corresponded to data in traditional programming. A class is similar to an abstract data type, although for object-oriented programs, the data typing process is not revealed outside the class. A class is an extension of the idea of the struct found in traditional programming language. It is a way of implementing a data type and associated functions and operations.

A method is similar to a procedure because both have processing operations. Methods and data are different because procedures are seldom encapsulated with the data they manipulate.

In an object-oriented program, message passing replaces function calls as the primary method of control in object-oriented systems.

TABLE II

A COMPARISON OF TRADITIONAL AND OOP CONCEPTS

Object-oriented Programming Concepts	Traditional Programming Concepts
Properties	Data
Classes	Abstract data types
Methods	Procedures, functions,
Messages	Function call
Calls under system's control	Calls under programmer's control
Inheritance	No

Inheritance has no similarity corresponding to the traditional programming. Table II summarizes the contrasts between the object-oriented and conventional perspectives (Date, 1990).

The object-oriented programming paradigm differs from the traditions of procedural programming. Procedural programming focuses on data and procedures with no constraints on which procedures act on which data. Data are structured so that they can be acted upon procedure by a separate and changing sets of procedures. Both the structure of the data and the organization of the procedures are subject to change, each potentially invalidating the other. The languages and techniques of the procedural programmer are all built to support this procedural programming paradigm.

Several significant departures from the procedural approach drive the way object-oriented programs are constructed. First, programs are collections of only one basic entity, the object, which combines data with the procedures that act upon the methods. Second, unlike traditional programs, which use procedures to accomplish actions on a separate set of passive data, objects receive requests and interact by passing messages to each other. Third, the hierarchical organization of objects into classes allows data and methods in one base class to be inherited by the subclasses.

Relational DBMS vs OODBMS

The section presents the comparison of an object-oriented data model and a relational data model. Object-oriented data models offer all the traditional data model advantages, including persistence, sharing, consistency, and query.

Persistence

Creating objects that service the process that created them. The objects considered so far exist for the duration of a computing session. There is also a need for objects with a longer time span.

Sharing

Sharing means not only that existing programmers and users can share the data in the database, but also that new programmers and users can be developed to operate against that same stored data. In other words, it may be possible to satisfy the data requirements of new programmers and users without having to create any additional stored data.

Consistency

It should also be clear that if the given fact is represented by a single entry, then such an inconsistency can not occur. Alternatively, if the redundancy is not removed but is controlled, then the data model could guarantee that the data model is never inconsistent as seen by the user, by ensuring that any change made to either of the two entries is also applied to the other one automatically.

Associative retrieval through a query

In an object-oriented data model, the user can specify queries with a nested dot notation rather than using a join, In an object-oriented data model there is no need for many of the joins used in relational data models, as there joins often serve to recompose entities that were decomposed for data normalization.

An object-oriented data model concepts depend on several important features beyond those offered by object-oriented language. Object-oriented data models, differ from relational data models in several ways including (Kim, 1989) :

Complexity, highly interconnected object

Because inter-object references are stored directly in an object-oriented data model, an object's identity must be invariant over changes to its state, and time must be a qualifier of identity to allow references to specific states of an object or to the "most recent" state.

Navigational model

An object-oriented data model is a navigational model of computation. A relational data model is based on a mathematical theory. A relational data model was never really designed to permit for the nested structure and views of a design. The advantage to navigation, especially when using large, complex applications such as those used in engineering design, and it is easier and much more natural to weave the way through objects that model the real world rather than tables, tuples, and records.

Type extensibility

A relational data model has relations as its only type. The operations on all relations are restricted to retrieve the data and store the data. However, an object-oriented data model includes a rich set of types, such as strong typing, abstract data types, procedures, and inheritance. An object-oriented data model supports type extensibility. Each object is associated with a class. User defined objects, or classes, are at the same level as the built-in types. The interface to each object is customized to the object. A user can accomplish reuse automatically by creating a subclass and overriding some of its properties and methods.

Strong type checking Data

An object-oriented data model may perform type checking at the database rather than the programming language level. Moreover, messages to manipulate data can be sent directly to data in the database rather than first moving data to the memory space of the programming language. This makes programs easier and more efficient.

Composite Objects

A composite object is a collection of other objects. Composites can be locked, stored, retrieved, and moved as a whole. In relational data model, there is no way to identify composite objects. The similar mechanism is the "view", which can join together multiple relations to form a virtual table. However, programmers have to assemble composite objects. This requires writing many slow and complex queries.

Abstract data type

In an object-oriented data model, the properties and methods of an object are represented by a class definition. Each of these classes is defined by a data abstraction. By incorporating data abstractions at the level of the data model, it is possible to make changes to the way a data model class is implemented without any effect on other classes in the data model that make use of the abstraction.

Better suited to store Data

An object-oriented data model may be viewed as a component of the programming environment that is actively integrated with other tools in the environment. Object-oriented data models are better suited to storing programs, objects, and types than relational data models.

Support for long transactions

Object-oriented data models may include support for long transactions , complex objects, heterogeneous objects, and changes management, like version control, action triggers, exception handling, type changes.

Read and Write locking

The relational data model and object-oriented data model provide a similar concurrency control scheme. Each provides read and write locking at a record or an object. Each also provides write aggregate locking at a relation or a segment. However, there is a difference between the two data models when it comes to locking information contained in an ownership hierarchy. In a relational data model, because hierarchies are represented across several relations, the application needs to lock each record specifically in each relation to lock the hierarchy. However, in an object-oriented data model, the programmers can lock pre-defined containment hierarchy with a single operation.

Conclusion

Object-oriented data models differ significantly in functionality from relational data models. Relational data models are based on deriving a virtual structure at run time based on values from sets of data stored in tables. Object-oriented data models contain predefined objects that do not need to be derived at run time. In a relational data model, views are constructed by selecting data from multiple tables and loading them into a virtual table. In an object-oriented data model, views are obtained by transversing pointers from object to object.

An object-oriented data model plays a very active role, whereas a relational data model is passive. While the relational data model primarily supports the ability to insert or delete records, the object-oriented data model offers the ability to combine methods within objects; thus it permitting the data model to combine many of the operations that must be left to the application with a relational data model.

CHAPTER IV

OODB IMPLEMENTATION

An OOP Extended to a Data Model

Relational DBMSs have a greater foundation in theory than either network or hierarchical systems motivating an object-oriented approach to relational database system implementation (Baroody 1981).

Objects extension to relational database are usually implemented as objects flattened into tables or with tables containing pointers to the objects. Object-oriented features are being added to database to increase their support for complex data types and complex data access (Dawson, 1989).

The extended relational approach starts with a relational model of data and query language and extends them in various ways to allow the modeling and manipulation of additional semantic relationships and database facilities.

The external level focuses on the fundamental data structure. The conceptual level contains generic, DBMS-independent tables and maps external-level object structures into tables and domains. The internal level is the data definition language of the target data models and contains the actual data model commands that create tables, attributes and indexes. (Baroody, 1981) (Chen, 1976)

Object

In the external level, an object is anything that both exists and has an identity. Objects such as apple, green, and Oklahoma belong to the object classes fruit, color, and state, respectively. An object is a data of an object class described by attributes or fields. In the conceptual level, each object class maps directly to a table. All object fields become attributes of tables. Each object has a unique ID; all references to objects are made by the ID. The stability of object IDs is particularly important for relationships since they refer to objects. Domains both ensure consistent decisions on attribute length and prevent operations on inconsistent entities. It does not make sense to add a name to a year. The concept of domain is similar to strong typing in a programming language.

In order to design an object-oriented data model for water resource management. We use two fundamental types of relationships: generalization and aggregation relationships.

Generalization

In the external level, a generalization refers to the ability to organize objects in an IS-A hierarchy. Generalization can have an arbitrary number of levels. For example (Figure 3) for the top generation, Water is the superclass; River, Lake, and Sea are subclasses. The superclass stores general properties like PH, temperature (temp). The subclasses store properties particular to each type of Water. In the conceptual level, a generalization relationship has one superclass table and multiple subclass tables. In the conceptual level identifies candidate keys. We choose one of the candidate keys to be the primary key. The object ID usually is the primary key for object tables. IDs are the primary key even though they have no inherent meaning to the user. The primary key must be unique and non-null. In the internal level is the data definition language of the target data model. The level contains the actual data model commands that create tables, attributes, and index.

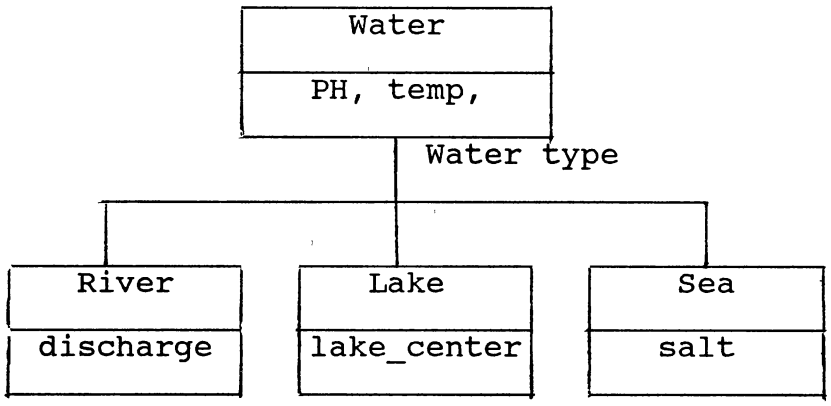


Figure 3. Generalization Relationship

Aggregation

Aggregate type constructors include set, multiset, list and tuple. The aggregation relationship is an object-oriented context is simply that a class consists of a set of objects. In the external level, aggregation is an assembly-component or IS-PART-OF relationship. Aggregation combines low-level objects into composite objects. Aggregation may be both multilevel and recursive; like a data structure it may act recursively and refer to itself. For example (figure 4), temperature is a component of characters, PH is a component of characters. Characters is an assembly and temp and PH are components. Aggregation often exhibit existence dependency. In the conceptual level, many-to-many relationships by necessity map to distinct tables. This is a consequence of normal form. One-to-one and one-to-many relationships may be mapped to distinct tables or merged with a participating object. Our handling of one-to-one and one-to-many aggregations depends on the context. We merge existence dependent aggregations with an object table to simplify integrity enforcement.

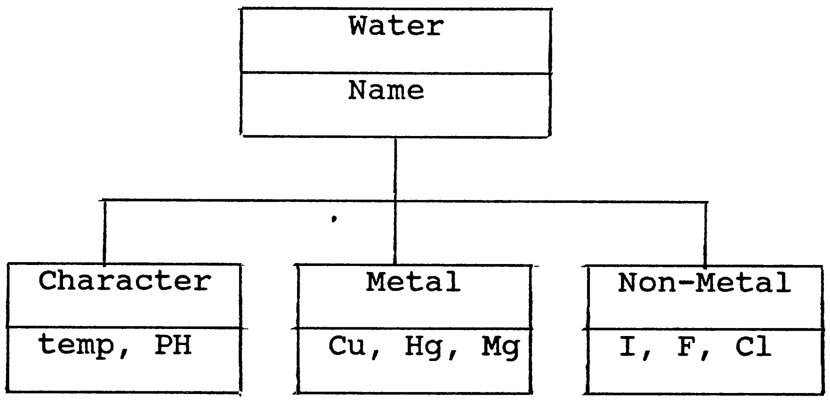


Figure 4. Aggregations Relationship

An OODB Design and Implementation

This section presents a description of the object-oriented data model design and implementation. We will develop a data definition language (DDL), data manipulation language (DML), and a query language. Using the data definition language, the classes must be in a format that the computer can translate into physical storage characteristics for the data. We will use the data definition language compiler program to translate a DDL text into a DDL C++ language. To update and retrieve data, we need a data manipulation language that can process data. The DML needs to insert, delete, retrieve, and update data. The Query utility program allows the user to extract data from the data model by specifying a class name, a list of data to extract from the class, and a set of select criteria. In developing and designing the object-oriented data model, the following steps will be taken. We describe our design of an object-oriented data model for water resource database below.

Step 1: Identify the basis for the object-oriented data model requirements.

Step 2: Define and describe objects: an object represents an entity in the real world. Objects may be system-defined objects or user-defined objects.

Step 3: Define classes: each class has a set of methods that it defines and a list of objects to which its instances pass messages.

Step 4: The data definition language (DDL) describes the objects to the application program. This step defines data types and specifies their combined properties and operations.

Step 5: The data manipulation language (DML) provides the four fundamental operations of insert, delete, retrieve, and update.

Step 6: A query utility program allows the user to extract data from the database by specifying a class name, a list of data elements to extract from the class, and a set of select criteria.

Implementation Steps

Step 1: Identify the basis for the object-oriented data model requirements. The candidate classes and objects are usually derived from perceptible things: people, events, and interactions. We may range from informal to formal. It may be sufficient to simply list the names of classes and objects, using meaningful names that imply their semantics. We must arrange those classes into a meaningful hierarchy.

Grady Booch created the grammatical approach. The procedure creates a list of the key nouns, noun phrases, and verbs from the data model requirements. Such a list may serve as an approximation to the problem-domain entities. He suggests using the nouns as potential identifiers of the classes of objects. Verbs, on the other hand, identify methods. The resulting list of nouns, noun phrases, and verbs is then used to begin the design process (Booch, 1983).

The grammatical approach starts with a statement of the model requirements and description of the solution, as shown in the following example:

Develop a water resource database system. The water resource database system allow the users to create water, river, and riverbed classes. Moreover, water can be <u>tested.</u> river can be <u>inserted,</u> <u>deleted,</u> <u>updated,</u> and <u>selected.</u> riverbed can be <u>surveyed.</u> Then, these classes may be define as follows.

```
class      water
property   name, date, PH, temp, hard
method     test, test1

class      river
property   discharge
method     insert, delete, retrieve, update, display
```

class	riverbed
property	ll, lr, ul, ur
method	riverbed, circum, diagonal

Step 2: Define and describe objects: an object represents an entity in the real world. Objects may be system-defined objects or user-defined objects. The system-defined object types are those predefined by the system. The user-defined types are the application specific types defined by the user.

Objects represent entities and concepts form the application domain being modeled. They are unique entities in the data model with their own identity and existence, and they can be referred to regardless of their attribute values.

Objects are described by their behavior, and can be accessed and manipulated only by means of methods. As long as the semantics of the methods remain the same, the data model can be physically as well as logically reorganized without affecting application programs. This provides a high degree of data independence.

A simple object is an abstraction of the notion of a variable-- both are responsible for holding data values such as strings or integers. A complex object may itself, in turn, aggregate simple and complex objects. Complex objects may also have a set of occurrences. A complex object may be thought as an abstraction of the notion of a struct or a record in traditional programming languages. It allows us to use nested structure too (Jacob, 1988).

A complex object may also be thought of as a tree. The object itself is the parent node, and the objects that it aggregates are the children of the parent node. Each of the children node objects, which are simple, is a leaf node. Each of those that is complex is itself a parent node. The nodes of a set occurrence are siblings (Dawson, 1989). The model class definition code fragment (Fig 5) below illustrates the hierarchical structure of an example hydraulic object's contents :

```
class water {

 // properties
     char    : name;                      // station name
     date    : date;                      // test date
     float   : spec;                      // specific code
     float   : ph;                        // PH standard
     float   : temp;                      // temperature
     float   : hard;                      // hardness
     int     : depth[10];                 // river depth
     riverbed : trap {                    // Riverbed shape
                 int :ll;                 // lower left
                 int :lr;                 // lower right
                 int :ul;                 // upper left
                 int :ur;                 // upper right
             }

  // methods
     test1(float ph);                     // test PH
     test2(float temp);                   // test temp
}
```

Fig 5. Class Definition Code Fragment

The station_name (name), test date (date), specific code (spec), ph standard (ph), hardness (hard), temperature (temp), river depth (depth), and trapezoid (trap) objects are aggregated by the complex parent water object. The trapezoid object is a complex object; the others are simple objects. The trapezoid object is parent to a set of identically structured complex objects, composed of the simple objects (lower_left, lower_right, upper_left, and upper_right).

All system-defined objects, both simple (integer, float, string..) and complex (array, list, set, tree..) can be used in the formation of user-defined objects. Once a user-defined object has been accepted by the object manager, it may be used indistinguishable from system-defined objects in the formation of further user-defined objects.

Any complex object is an aggregate of other objects, any complex object type is an aggregate of other types. Thus the complex data type water is the aggregate of the name, date, spec, ph, hard, temp, depth, and trap types. Note that the river type definition is the user-defined object that is made available for reuse in later user-defined type definitions; its component object types are not (Kim, 1989).

Step 3: Define classes: each class has a set of methods that it defines and a list of objects to which its data pass messages. The step may establish the classes and objects identified from step 2. The designer acts as a separate outsider, viewing each class from the view of its interface so as to identify the things. S/he manipulate do to each instance of a class and manipulate the things that each object can do to another object.

After the objects are described, we can gather common object abstractions together in a class definition. For example, we might define a class called water. The water class for storing name, date, spec, ph, hard, temp, depth, and trap. The river subclass for storing discharge instances are then created.

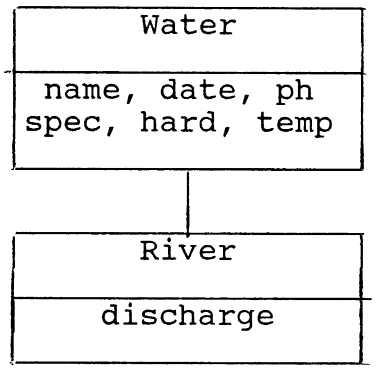


Figure 5. The IS-A Hierarchy in Water Resource Database

The class provides a variety of base types and type constructors for defining schema types, some of which are used in Figure 6. Predefined base types include integers, floating point numbers, and character strings.

Inheritance increases flexibility by permitting the addition of new class of data types without having to modify the existing code. As an example, consider the following class, river, derived from class water.

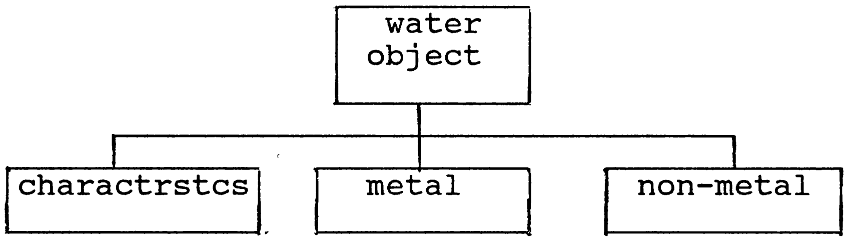
The subclass river inherits all of the properties and methods of the class water, and in addition it has discharge property.

```
// The parent class water has the properties (name, date,
// spec, Ph, hard, temp, and depth) and methods (test,
// test1).
//
//
 class Water
    {
     // properties
           char    *name;
           char    *date;
           float   spec;
           float   ph;
           float   hard;
           float   temp;
           int     depth[10];
           riverbed  trap;
      // methods
           int test (float ph, float hard);
           int test1(float ph, float date);
     }
```

```
// subclass river inherits all the properties from class
// water
class river  : public water
    {
      // properties
             float   discharge;
             River   *next;
      // methods
             insert();
             delete();
             update();
             select();
     }
```

Figure 7. An Example of Water Resource Database

A water object, which is a member of the water class, consists of basic characteristics, metal-element and nonmetal-element. A metal-element object that belongs to the metal-element class has Na, Cu, Mg, and Hg as its components.



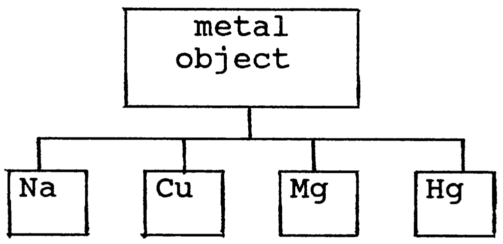


Figure 8. IS-PART-OF Hierarchy in Water Resource Database

Step 4: The data definition language (DDL) describes the objects to the application program. This step defines data types and specifies their combined properties and methods. DDL compiler compiles the user-defined objects into the database, where they supplement the system-defined objects.

It is used to define a schema or data model for an object-oriented data model. DDL is the class definition language which extends the hierarchy of system defined classes in an object-oriented data model, and defines the relationships among the data objects and class. DDL is used to define an abstract interface to the properties of new data types which may be instantiated and the operations which may be performed on these properties. Each class definition code segment enclosed between "create" and "end" contains a data definition statement, which defines the name of the class, its superclass, and its associated properties and its methods. It is a sample DDL definition:

```
create class water
      Properties = {
             char   *name;
             char   date;
             float  spec;
             float  PH;
             float  hard;
             float  temp;
             int    depth[10];
             riverbed  trap;
      }
      Methods = {
         int  test(float ph, float hard);
         int  test1(float ph, float date);
      }
end Water;


create class river
        Superclass = { Water }
        Properties = {
               float discharge;
               River *next;
        }
        Methods = {
           void     insert();
           void     delete();
           void     update();
           void     select();
        }
end River;


create class riverbed
       Properties = {
              int ll;
              int lr;
              int ul;
              int ur;
       }
       Methods = {
              int area(int ll, int lr, int ul, int ur);
              int circum(int ll, int lr, int ul, int ur);
              int diagonal( int ll, int lr, int ul, int ur);
       }
end Riverbed;
```

Figure 9. A Data Definition Language

A method consists of the word "Method", followed by the method-name as defined in the definition of the class that defined the operation. A method is similar to the definition of a programming language's function or procedure. In a data definition language, the following built-in data types are provided:

1. integers, boolean
2. floating point
3. fixed-length string
4. unbounded varying length arrays of fixed types with an arbitrary number of dimensions. e.g., text, image, or sound.
5. Procedure used to represent shared complex objects and to support multiple representations of data.

```
class riverbed {
  int area()     { XXXX }
  int circum()   { XXXX }
  int diagonal() { #### }
}
```

Step 5: A data manipulate language (DML) includes facilities to express queries and updates against a given database (insert, delete, update).

The DML of object-oriented data model must allow users to insert, delete and update appropriate objects of a class by sending an appropriate message to the class. It must allow users to fetch objects in several ways. First is a navigational fetch of a single object keyed on its unique identifier or of a collection of objects rooted at a user-specified object. Second, it permits a declarative fetch of a set of objects that specifies user-specified search conditions. Third, DML must permit a declarative deletion or modification of a set of objects that satisfy user specified search conditions.

In a relational data model, an insert operation encapsulates these actions: the creation of a tuple, and the insertion of the tuple in a certain table. In an object-oriented data model, we can extend the semantics of the SQL form and insert (delete, and update) into clauses to permit any aggregate expression. We employ a DML which is similar to the DML in the relational database.

For example, we may use following statement to add the value 3 to the "depth[4]" which is one of the properties of the object in the river class, p.

```
insert
into   p (depth[5])
Values ("3")
```

For another example, the following "update" command is used to modify the value of ph, discharge, and set depth to "5,4,3,2,6,7,8,0,4", respectively.

```
modify p
set    ph   = ph * 1.1
       disch = 400
       depth = "43267804";
where  ph = 7;
```

The "delete" command is used to delete objects, for example,

```
delete
from p
where depth[3] = 2;
```

Step 6: A query utility program permits the user to extract data from the database by specifying a class name, a list of data elements to extract from the class, and a set of select criteria. The query language provides facilities for selecting complex object structures, and it can be extended through the addition of ADT functions and operators, procedures and functions for manipulating schema types, and generic set functions.

For example, the "select" command herein is used to select name, date, the area value of trap, and to get the diagonal value of the trap.

```
select name, date, area(trap), diag(area)
from p
where disch  = 80;
```

In a relational data model, views are constructed by selecting data from multiple tables and loading them into a virtual table. In an object-oriented data model, a view is obtained by transversing pointers from object to object.
A complex query, as it involves a number of classes, brings out many of the same issues that complicate joins of relations in relational data model. The semantics of queries in object-oriented data models make it possible to eliminate many of the permutations of classes that may not be eliminated in evaluating relational queries. For example,

Relational Model

```
  create table Depth (
  Riverbed_depth : char(5),
  index : integer,
  value : integer);
  insert into Depth
  values('D1',0, 3);
  insert into Depth
  values('D1',1, 4);
            :
            :
            :
  insert into Depth
  values('D1',9, 8);
  select values into y
  from Depth
  where Riverbed_depth = D1 and index = 8;
```

Object-oriented Model

```
  Define :
  int  Riverbed_depth[10];
  D1[0] = 3;
  D1[1] = 4;
      :
  D1[9] = 8;
  Y = D1[8];
```

In this example, SQL is wordy, and for more complex data types, is difficult to write, hard to understand, and nearly impossible to modify and maintain.

In our experiment, Insert, Modify, Delete, and Select are implement by four individual functions written in C++.

Analysis

The following problems seek to use the relational data models as generic design data models; however, object-oriented data models should resolve the problems.

As is well known, the relational data model expresses the semantics of complex objects with difficulty because it is only a table model for data storage. Object-oriented programming can model real-world entities appropriate to the user's needs. Assume we need a data type of one dimensional array (river_depth) in the water resource database. One dimensional array is an easy-understood data structures. Most applications frequently use it. Nevertheless, there is no direct way to store an one dimensional array data type in the relational data model. We need to translate a one dimensional array into a table. In relational data models, each data structure must be mapped into a table by the database designer. To store the one dimensional array data type, we need a table called depth with three columns: the riverdepth, index, and the actual value. The Index column is required because rows in a relational table have no implied ordering. These columns of the relational data model take extra space on the memory. More space requires additional time to copy and transfer. SQL is wordy, for more complex data types, is difficult to write and modify, hard to understand and maintain. Many frequently used data structures, e.g., matrixs, sets, link lists, and trees cannot be directly represented as tables. The relational model does not directly support these complex data structure. They need many expensive access translations.

Name	date	spec	ph	hard	disch	temp	depth	trap
ILLIO55	02/07/1986	120	7	98	180	5	A1	r1
ILLIO55	04/21/1986	220	8	109	472	26	A2	r2
......								

water (name, date, spec, ph, hard, disch, temp, depth, trap)

riverdepth	index	value
A1	0	3
A1	1	4
A1	2	0
A1	3	1
A1	4	3
A1	5	2
A1	6	5
A1	7	9
A2	0	1
A2	1	8
..		

Depth(riverdepth,index,value)

trap	point	X	Y
r1	ll	0	0
r1	lr	3	0
r1	ul	0	5
r1	ur	3	5
r2	ll	1	1
r2	lr	5	1
r2	ul	2	4
r2	ur	5	4
....			

Trap (trap, point, x, y)

Figure 10, Water Resource Database Stored in Relational Tables

Class River

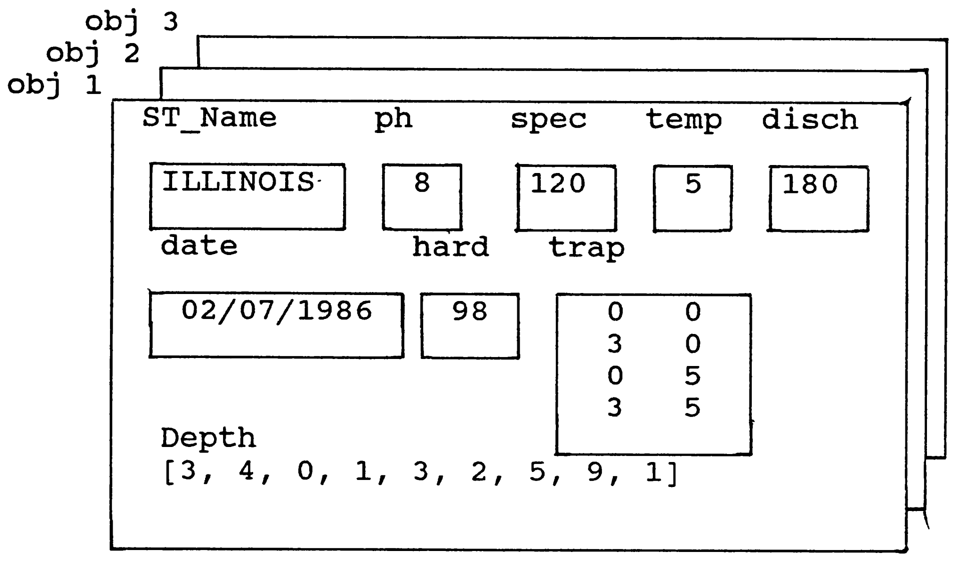


Figure 11. Water Resource database stored in Object-Oriented Database

Figure 10 and Figure 11 illustrate the differences between tables and objects, using the example of water resource data stored in a relational data model and an object-oriented data model. In a relational data model, the three base tables T1, T2, and T3 are indexed on the primary key, then all the tuples can be directly accessed using the primary key. For each manipulating operations, the time complexity in a B-tree is $(\log_m T1 + \log_m T2 + \log_m T3)$, where m is the order of a B-tree and Ti indicate the numbers of tuples stored in the table Ti. In object-oriented data model, table are indexed on the primary key, then all the objects can be directly accessed using the primary key. The time complexity in a B-tree is $\log_m C1$, where m is the order of a B-tree and C1 indicate the number of objects stored in the Class C1. With a simple example, notice how an object can contain other objects to any level of nesting, thus providing flexibility in defining new object types. As illustrated in the example, objects make writing applications easier, because all of the data for an entity are located in one place. In an object-oriented database, the programmers do not need to search through multiple files by pointers to determine where the river's depth is stored.

A relational data model has only a limited set of built-in data types (e.g., integer, string, date), and a limited set of built-in operations on these data types (e.g., select, project, retrieve, and store). The object-oriented data model provides richer data types and operations for defining and managing the complex software.

A relational data model designer can connect basic data types linearly to create complex data types, like as joining fields into records, for example,

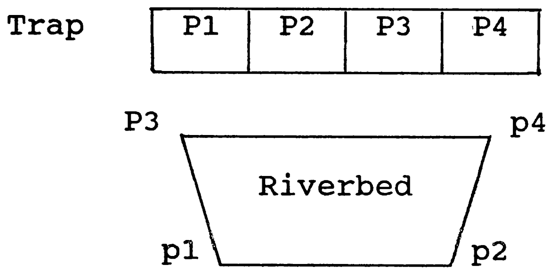


Figure 12. Complex Data Type Riverbed

However, we can not add new operations for the new complex data type, the operations on it are limited to those defined for the basic data types. For example, riverbed is represented by four points p1, p2, p3, and p4, marking the lower-left, lower-right, upper-left, and upper-right corners of the Riverbed, respectively. In an object-oriented data model, we can create a method area(), applied to a particular Riverbed which computes the area of the trapezoid. In a relational data model, it cannot provide an operation to computer the area. The definition of the data type Riverbed demonstrates how user-defined data types can be provided like in built-in data types. Moreover, each object in an object-oriented data model is a member of a class, which determines the object's structure and defines what operations can be performed on it. New classes are created from existing classes through the technique of subclassing and overriding of inherited properties and methods. This results in complex data types and operations, which may then be treated as if they are the built-in types.

Object inheritance increases extendibility and reusability and saves the cost of maintenance. Classes provide not only modularity and information hiding but also reusability enhanced by inheritance and polymorphism. A relational data model designer can reuse code by copying or editing. An object-oriented data model designer can accomplish this automatically by creating a subclass and overriding some of its properties and methods.

Subclasses can duplicate or inherit the properties and methods of existing classes. The subclass inherits common properties and methods from individual classes and permits programmers to be factored into ever more refined modules. As shown in figure 15, the class lake can be created as a subclass of the class water to further describe lake which are responsible for a group. Class lake inherits all the properties and methods from its parent class, water, without duplicating its code. Differences between water and lake are expressed in the lake's methods and data. New lake data structure like lake_center and lake_circum are declared. Class lake inherits the methods such as Test() and Test1() from its parent class water. Thus, a lake is like a water, with some differences. As this example illustrates, a programmer is not required to build from scratch.

```
Class water
{
  // properties :
     char   *name;
     char   *date;
     float  spec;
     float  ph;
     float  hard;
     float  temp;
     int    depth[10];
     riverbed   trap;

  //  methods :
     int  Test(float ph, float hard);
     int  Test1(float temp, float hard);
}
```

Figure 13. Class Water.

```
// Class river inherits the properties date, ph, hard, and
// temp, and the method Test(), Test1() from class water
class river : water
{
// New properties discharge are declared

     float   disch;
     river   *next;

// Class river inherits methods test() and test1() from its
// parent water
// New river methods like insert(), delete(), update() and
// select() are added.

     void  insert();
     void  delete();
     void  update();
     void  select();
 }
```

Figure 14. Class River.

```
class lake : water   // Class Lake inherits from class Water
{
// Inherits all the data structures from its parent class //
water.  Without duplicating its code.
// new property depth are declared.

     int lake_center;
     int lake_circum;
 // Class lake inherits methods test() and test1() from its
 // parent class water.

     int test2(int depth, char *date);
 }
```

Figure 15. Class Lake.

Traditional relational data models are closed systems to which the programmer is not permitted to make any changes. Schema modification or the restructing of a database will become an issue as upgrades or new versions of object-oriented data models are required. This has always been a problem in the relational model when large amounts of data must be converted to a new format. With an object-oriented data model, the problem could be minimized by subclassing.

Conclusion

Relational database has strong mathematical foundation. It is also understandable, data independent, and easy to manage business-like information. For the scientific and engineering applications, non-record data types, such as arrays, link lists, trees, sets, and complex graphics are need. Then, object-oriented data model can provide complex computing ability over than relational database. The object-oriented data model can provide richer types and operations for managing the complex objects.

CHAPTER V

SUMMARY, CONCLUSIONS, AND SUGGESTIONS FOR FUTURE WORK

The relational database was created to provide a simple abstraction that allowed the representation of large access of data using a small set of principles. Similarly, the object-oriented database was designed to permit creation and description of complex data structures in a coherent and uniform way.

For the business application, the relational database can solve most business data processing problems. However, for the engineering applications, a traditional relational database is not appropriate. The major design goals of the object-oriented database (OODB) are :

First, OODB proposed to support complex data types, engineering data, in contrast to business data, are more complex and dynamic.

The second goal is to allow new data type, new applications and new access methods to be included in the OODB.

The third goal is to increase relational database that match real-world applications and reduce the normalization problems often appeared in the relational design.

The fifth goal is to support code reuse, code maintenance, and modularity.

The following improvements are suggested for future work :

First, expect some languages to develop their own native interfaces to object-oriented database.

Second, provide a graphic utility for the designer of a database using the class types and relationships of object-oriented data model.

Third, choose extended relational database or object-oriented database to solve the complex computing needs in the future.

Fourth, make as few changes to the relational model as possible. Many users in the business data processing world will become familiar with relational concepts and this framework be preserved if possible.

Fifth, extend the query language to allow more operations to be specified declarative would allow better optimization and use of indexes. Making use of instance variable typing in complied methods to allow earlier binding may reduce execution time.

In this thesis, we report our design and experiment of an object-oriented database for water resource management. From this database, we found that the object-oriented paradigm provides better representations of objects, and better reusability and extendibility of software than a relational database. The major disadvantages of our object-oriented database are expenses of the start-up costs and slower speed than the relational database.

BIBLIOGRAPHY

Agrawal, R. (May 1987), "Object Database and Environment: The Language and the Data Model," Communications of ACM, pp. 26-48.

Andrews, T. and Harris, C. (October 1987), "Combining Language and Database Advances in an Object-Oriented Development Environment," OOPSLA '87 Proceedings, pp. 430-440.

Banerjee, J. and Kim, W. (May 1987), "Semantics and Implementation of Schema Evolution in Object-Oriented Database," Communications of ACM, pp. 311-322.

Baroody, A. J. and Dewitt, D. J. (December 1981), "An Object-Oriented Approach to Database System Implementation," ACM Transactions on Database Systems, Vol. 6, No. 4, pp. 576- 601.

Bernstein, P. A. (March 1987), "Database System Support for Software Engineering - An Extended Abstract," Proc. 9th International Conf. on Software Engineering, pp. 166-178.

Bloom, T. and Zdonik, S. B. (October 1987), "Issues in the Design of Object-Oriented Database Programming Languages," OOPSLA '87 Proceedings, pp. 441-451.

Booch, G. (March/April 1987), "Object-Oriented Design," Ada Letters, Vol.1, No, 3.

Cargill, T. A. (September 1986), "A Case Study in Object-Oriented Programming," OOPSLA '86 Proceedings, pp. 350-360.

Chen, P. (March 1976), "The Entity-Relationship Model - Toward a Unified View of Date," ACM Transactions of Database Systems, Vol. 1, No. 1, pp. 9-36.

Chennho, K. (November 1990), "Object Subclass Hierarchy in SQL : A Simple Approach," Communications of the ACM, pp. 99- 108.

Danforth, S. and Tomlinson, C. (March 1988), "Type theories and Object-Oriented Programming," ACM Computing Surveys, Vol. 20, No. 1, pp. 29-71.

Date, C. J. (1990), "An Introduction to Database Systems," Addison-Wesley, Reading, Mass.

Dawson, J. (September 1989), "A Family of Models," Byte, pp. 277-286.

Duhl, J. and Damon, C. (September 1989), "A Performance Comparison of Object and Relational Databases Using the Sun Benchmark," OOPSLA '88 Proceedings, pp. 153-163.

Fishman, D. H. and Beech, H. P. (January 1987), "Iris: An Object-Oriented Database Management System," ACM Transactions on Office Information Systems, Vol. 5, No. 1, pp. 48-69.

Garza, J. F. and Kim, W. (March 1988), "Transaction Management in an Object-Oriented Database System," ACM Transactions on Office Information Systems, pp. 37-35.

Goldberg, A. (1983), Smalltalk-80: The Language and Its Implementation. Reading, Mass.: Addison-Wesley.

Hornick, M. F. and Zdonik, S. B. (January 1987), "A Shared, Segmented Memory System for an Object-Oriented Database," ACM Transactions on Office Information Systems, Vol. 5, No. 1, pp. 70-95.

Jacob, S. (March 1988), "Object-Oriented Programming and Databases," Dr. Dobb's Journal, pp. 18-34.

Kemper, A. (May 1987), "An Object-Oriented Database System for Engineering Applications," Communications of ACM, pp. 299-310.

Kim, W. and Ballou, N. (September 1988), "Integrating an Object-Oriented Programming System with a Database System," OOPSLA '88 Proceedings, pp. 142-152.

Kim, W. (1989), "Object-Oriented Concepts, Databases, and Applications," Reading, MA: Addison-Wesley.

Liu, L. C. and Horowitz, E. (January 1988), "Object Database Support for a Software Project Management Environment," Communications of ACM, pp. 85-96.

Maier, D. and Stein, J. (1986), "Development of an Object-Oriented DBMS," OOPSLA '86 Proceedings, pp. 472-482.

Meyer, B. (February 1987), "Eiffel: Programming for reusability and extendibility," SIGPLAN, Notices, Vol 22, No 2, pp. 85-94.

Mayer, B. (1988), "Object-Oriented Software Construction," Englewood Cliffs, N.J.:Prentice Hall.

Penney, D. J. and Stein, J. (October 1987), "Class Modification in the GemStone Object-Oriented DBMS," OOPSLA '87 Proceedings, pp. 111-117.

Rentsch, T. (September 1982), "Object-Oriented Programming," SIGPLAN Notice, pp. 51-57.

Rubenstein, M. S. and Kubicar, R. G. (May 1987), "Benchmarking Simple Database Operations," Communications of ACM, pp. 387-394.

Sauid, A. (1988), "Object-Oriented Database Programming," Springer-Verlag.

Sally, S. and Meller, S. J. (1988), "Object-Oriented Systems Analysis," Englewood Cliffs, N.J.:Prentice Hall.

Shriver, B. and Wengner, P. (1988), "Research Directions in Object-Oriented Programming," The MIT Press.

Willian, J. (November 1990), "An Object-Oriented Relational Database," Communication of the ACM, pp. 99-108.

Woelk, D. and Kim, W. (January 1986), "An Object-Oriented Approach to Multimedia Databases," Communications of ACM, pp. 311-325.

Wu. C. T. (March/April 1990), "Benefits of Object-oriented Programming in Implementing Visual Database Interface," Journal of Object-Oriented Programming, pp. 8-16.

Zdonik, S. (1990), "Readings in Object-Oriented Database Systems," San Mateo, CA : Morgan Kaufmann.

APPENDIXES

APPENDIX A

GLOSSARY

The terminology below must be defined prior to discussing the concepts of object-oriented programming (Rentsch, 82).

Abstract Data Types --

A set of data structures or data types is defined in terms of the structure's properties and the operations executed on these structures. In object-oriented programming, object types are similar to abstract data types.

C++ --

An object-oriented superset of the C language written by Bjarne Stroustrup at AT&T's Laboratories. The term C++, means "better than C."

Class --

The description of a set of nearly identical objects that share common methods and general characteristics.

IS-A hierarchy --

The IS-A hierarchy where a class has a subclass associated with it. The subclass inherits all of the instance variables and methods associated with its superclass.

IS-PART-OF hierarchy --

The IS-PART-OF hierarchy where an object of a class is considered to be the aggregation of a set of objects, each of which belongs to some class.

Method --

The function or procedure that implements the response when a message is sent to an object. Methods determine how an object will respond to a message that it receives.

Modularity --

Program construction in modules, blocks or units that are combined to build complete programs. Ideally, the redesign or reimplementation of a unit or module can be accomplished without affecting the operation of the rest of the program or system.

Multiple inheritance --

The ability of subclasses to inherit instance variables and methods from more than one class. It is useful in building composite behavior from more than one branch of a class hierarchy.

Object --

The original element in object-oriented programming is an object. Objects are entities that encapsulate within themselves both the data describing the object and the instructions for operating on that data.

Object identity --

Properties of an object that remains invariant across all possible modifications of its state. Can be used to point to the object.

Object-oriented database --

A database that allows data to be stored as objects.

Polymorphism --

The property of sharing a single action (and action name) by an object hierarchy, but with each object in the hierarchy implementing the action.

APPENDIX B

SAMPLE QUERY FROM THE COMPUTER PROGRAM

The select statement -- The search is carried out by the SELECT statement. Aggregate functions are also supported. Some examples of basic queries on object-oriented database whose schema are given below:

Q1 : Get stat_name, data and PH with PH = 7.

```
select name, date, ph
from p
where ph = 7;
```

Q2 : Select river_bed[1], river_bed[3] for all objects.

```
select array[1], array[3]
from p
where temp = 50
```

Q3 : Calculate the area, diagonal and circumference for each trapezium where discharge = 400.

```
select area(trap), diag(trap), circ(trap)
from p
where disch = 400
```

The insert statement -- The insertion adds new objects to a class, p.

Q4 :
```
insert
into p (name, date, ph, hard, depth[4]);
values ("ILLION55", "08/06/1980", "6.89", "2.7", "3");
```

The delete statement deletes qualified objects from a class. The objects are specified by a search condition.

Q5 :
```
delete
from p
where depth[4] = 5;
```

The modify statement updates objects values in existing class. The updating of a class can executed by providing the values to be updated or by using an expression to build these values.

```
Q6 : modify p
     set ph = ph * 1.1,
     disch = 400,
     depth = "4, 3, 2, 6, 7, 8, 0, 4";
     where disch = 400;
```

APPENDIX C

PROGRAM LISTING

```
/*********************************************************/
/*                                                       */
/*          Object-oriented Database Program             */
/*                                                       */
/*********************************************************/
/*                                                       */
/*         Author :  HuiChen Nee                         */
/*           Date :  December, 1990                      */
/*          Class :  COMSC 5000 - Thesis                 */
/*        Adviser :  Dr. G. E. Hedrick                   */
/*                                                       */
/*********************************************************/
/*                                                       */
/*  This program is for modeling an object-oriented data */
/*  model.  The object-oriented data model is written    */
/*  using an object-oriented programming language (Turbo */
/*  C++).  The data model is defined, manipulated,       */
/*  queried based on C++ and support data encapsulation  */
/*  and inheritance.                                     */
/*                                                       */
/*  The data manipulating language provides the four     */
/*  fundamental operations of insert, delete, select, and*/
/*  update.  A query utility allows the user to extract  */
/*  data from the database by specifying a class name, a */
/*  list of data elements to extract from the class, and */
/*  a set of select criteria.                            */
/*                                                       */
/*********************************************************/
```

```
#include <stdio.h>              // for printf, file i/o, etc
#include <string.h>             // for strcpy()
#include <stdlib.h>             // for itoa()
#include <conio.h>              // for getch()
#include <ctype.h>              // for isalpha() and isalnum()
#include <math.h>               // for sqrt()


FILE *in;
class region;
class water;
class river;


class region
{                               // properties
 friend class river;
 public:
   char  rect[12];
   void  basic(char rect[15]);
   int   a2i(char *str);
   char* md(char *ds, char *ss, int a, int b);
   int   diagonal(char rect[15]);
   int   area(char rect[15]);
   int   circum(char rect[15]);
};
int X, Y;


void region::basic(char rect[15])
{
   char rx[3], ry[3], lx[3], ly[3];

     md(lx,rect,1,2);
     md(ly,rect,3,2);
     md(rx,rect,5,2);
     md(ry,rect,7,2);
     X = a2i(rx) - a2i(lx);
     Y = a2i(ry) - a2i(ly);
}


int region::diagonal(char rect[15])
{
   int diag;

     basic(rect);
     diag = sqrt(X*X + Y*Y);
       return diag;
}


int region::area(char rect[15])
{
   int ar;

      basic(rect);
      ar = X * Y;
        return ar;
}


char* region::md(char *ds, char *ss, int a, int b)
{
   int k, n;

      n = strlen(ss);
      if (( 0< a && a <= n) && (0<b && b <= n) && (a + b -1 <
      {
 for (k = a; k <a +b; k++)
    ds[k-a] = ss[k -1];
    ds[b] = '\0';
      }
      else
 ds[0] = '\0';
      return ds;
}


int region::a2i(char *str)
{
   int s, flag3;

      if (*str == '-') {
 flag3 = (-1);
 str++;
      }
      else if( *str == '+') {
 flag3 = 1;
 str++;
      }
      else
 flag3 = 1;
 s = 0;
 while ( '0' <= *str && *str <= '9') {
    s = 10 * s + (*str) - '0';
    str++;
 }
       return (s * flag3);
}


int region::circum(char rect[15])
{
   int re;

      basic(rect);
      re = 2 * (X + Y);
```

```
     return re;
}


//  Class water is the base class of class river.
class water
{
 public:                                 // properties of class
   char name[10];                        // station name
   char ph[10];                          // PH (standard units)
   char hard[10];                        // hardness
   char date[15];                        // date
   char temp[10];                        // temperature
   char spec[10];                        // specific conductance
   char array[10];                       // one dimension array
   region *place;                        // class region
 // methods of class water
   char*  md(char *ds, char *ss, int a, int b);
   char*  gap(char d1, char d2[], char d3[]);
   char   gap1(char d1, char d2, char d3);
   char*  f2a(char *str, float c);
   char*  i2a(char *str, long int a);
   int    a2i(char *str);
   char*  d2s(char dat[15]);
   double a2f(char es[10]);
   void   test1(char ph[10]);          // test the PH standard
};


char* water::md(char *ds, char *ss, int a, int b)
{
   int k, n;
     n = strlen(ss);
     if (( 0< a && a <= n) && (0<b && b <= n) && (a + b -1
     {
 for (k = a; k <a +b; k++)
   ds[k-a] = ss[k -1];
   ds[b] = '\0';
     }
     else
       ds[0] = '\0';
     return ds;
}


char* water::gap(char d1, char d2[], char d3[])
{
   char result[15];
   double result1;
   long int result2;

     switch(d1) {
      case '+' : result1 = a2f(d2) + a2f(d3);
   result2 =result1;
   if (result2 == result1)
```

```
      i2a(result,result2);
    else f2a(result,result1);
    return(result);
       case '-' : result1 = a2f(d2) - a2f(d3);
    result2 = result1;
    if (result2 == result1)
      i2a(result,result2);
    else f2a(result,result1);
    return(result);
       case '*' : result1 = a2f(d2) * a2f(d3);
    result2 = result1;
    if (result2 == result1)
      i2a(result,result2);
    else f2a(result,result1);
    return(result);
       case '/' : result1 = a2f(d2) / a2f(d3);
    result2 = result1;
    if (result2 == result1)
      i2a(result,result2);
    else f2a(result,result1);
    return(result);
        default : puts("You have a wrong update format\n");
    break;
      }
}


char water::gap1(char d1, char d2, char d3)
{
   char result;
   int result1;

     switch(d1) {
       case '+' : result1 = (d2 - '0') + (d3 - '0');
    result = result1 + '0';
    return result;
       case '-' : result1 = (d2 - '0') - (d3 - '0');
    result = result1 + '0';
    return result;
       case '*' : result1 = (d2 - '0') * (d3 - '0');
    result = result1 + '0';
    return result;
       case '/' : result1 = (d2 - '0') / (d3 - '0');
    result = result1 + '0';
    return result;
       default  : puts("You have a wrong update format\n");
    break;
     }
}


char* water::d2s(char dat[15])
{
   char result[15], result1[5], result2[5];
```

```
        md(result,  dat, 7, 4);
        md(result1, dat, 4, 2);
        md(result2, dat, 1, 2);
        strcat(result, result2);
        strcat(result, result1);
        return (result);
}


char* water::f2a(char *str, float c)
{
     long int r1;
     float r;
     int  q, flag, count, i, b;

        count = 0;
          if (c <0 ){
   *(str++) = '-';
   count++;
   c = (-c);
          }
        flag = 0;
        b = c;
        for (i = 10000; i>= 1; i= i/10) {
  q = b/i;
  b = b % i;
     if (q !=0   || flag != 0){
        *(str++) = q + '0';
        count++;
        flag = 1;
     }
        }
        if (count == 0){
          *(str++) = '0';
          count++;
        }
        *str++ = '.';
        count++;
        b = c;
        r = 100000 *c - 100000 * b;
        r1 = r;
        for (i = 10000; i>= 1; i = i/10) {
          q = r1/i;
          r1 = r1 %i;
   if (( q == 0 && r1 == 0) || count > 7){
        *str = '\0';
        return str;
   }
   *(str++) = q+ '0';
   count++;
        }
        *str = '\0';
        return str;
}
```

```
char* water::i2a(char *str, long int a)
{
   long i;
   int q, flag;

     flag = 0;
     if (a < 0) {
       *(str++) = '-';
       a = (-a);
     }
     for (i = 100000; i>=1; i= i/10) {
       q = (int) (a/i);
       a = a % i;
       if (q != 0 || flag != 0) {
  *(str++) = q + '0';
  flag = 1;
       }
     }
     *str = '\0';
     return str;
}


int water::a2i(char *str)
{
   int s, flag3;

     if (*str == '-') {
       flag3 = (-1);
       str++;
     }
     else if( *str == '+') {
       flag3 = 1;
       str++;
     }
     else
       flag3 = 1;
       s = 0;
     while ( '0' <= *str && *str <= '9') {
       s = 10 * s + (*str) - '0';
       str++;
     }
     return (s * flag3);
}


double water::a2f(char es[10])
{
   float val, power;
   int i, sign;

     for (i = 0; es[i] == ' ' || es[i] == '\n' || es[i] ==
     i++)
     ;
     sign = 1;
     if (es[i] == '+' || es[i] == '-' )
       sign = (es[i++] == '+') ? 1 :-1;
     for (val = 0; es[i] >= '0' && es[i] <= '9'; i++)
       val  = 10 * val + es[i] - '0';
     if (es[i] == '.')
       i++;
     for (power = 1; es[i] >= '0' && es[i] <= '9'; i++) {
       val = 10 * val + es[i] - '0';
       power *= 10;
     }
     return (sign * val /power);
}


// the method test1() tests the PH value
void water::test1(char ph[10])
{
   int value;

     value = a2f(ph) - 7;
     if (value > 0)
       puts("Alkalinity");
     else if (value == 0)
       puts("Neutrality");
     else if (value < 0 )
       puts("Acidity");
}


// This class is derived from water and inherits all the
// members of the parent class
class river:water
{
public:
   char  disch[10];               // property discharge
   river *link;                   // point to next river
   void  inser();                 // method insert()
   void  delet();                 // method delete()
   void  modif();                 // method modify()
   void  selec();                 // method select()
   void  displ();                 // method display()
   void  read_input();            // method read_input()
   void  transfer(char word[50]); // method transfer()
   void  transfer1(char word[50]);// method transfer1()
   void  io(int i);               // method input/output()
   void  init();                  // method initial()
   void  read_file();             // method read_file()
   void  write_file();            // method write_file()
   void  menu();                  // method menu()
   char* mid(char *ds, char *ss, int a, int b);
};

char max[20][10], line[80], indata[10][12], tran[10][12];
char input[90], output[90], filenames[15];
```

```
int clock, clock1;
class river *record, *head, *tail, *potmp1, *tp1, *tp2;


void river::inser()
{
   char index[3], ai[10], names[10];
   int i, p, u, j;

     potmp1 = new river;              // create a new river
     tp2 = NULL;                      // sets node point to
     read_input();


// omitted the list of properties is equivalent to specifing
// list of all properties in the class
     if (clock ==1) {
       strcpy(indata[2],"1");
       strcpy(indata[3],"2");
       strcpy(indata[4],"3");
       strcpy(indata[5],"4");
       strcpy(indata[6],"5");
       strcpy(indata[7],"6");
       strcpy(indata[8],"7");
       strcpy(indata[9],"8");
       strcpy(indata[10],"9");
       clock = 9;
     }
     if(strcmp(indata[0],"into") != 0) {
       io(11);
       return;
     }
     read_file();
     for(i = 2; i< clock +1; i++){
       if ((j = strcmp(indata[i],"name")) == 0)
   strcpy(indata[i],"1");
       if ((j = strcmp(indata[i],"date")) == 0)
   strcpy(indata[i],"2");
       if ((j = strcmp(indata[i],"spec")) == 0)
   strcpy(indata[i],"3");
       if ((j = strcmp(indata[i],"ph")) == 0)
   strcpy(indata[i],"4");
       if ((j = strcmp(indata[i],"hard")) == 0)
   strcpy(indata[i],"5");
       if ((j = strcmp(indata[i],"disch"))  == 0)
   strcpy(indata[i],"7");
       if ((j = strcmp(indata[i],"temp"))  == 0)
   strcpy(indata[i],"8");
       if ((j = strcmp(indata[i],"place"))  == 0)
   strcpy(indata[i],"9");

       mid(ai, indata[i], 1, 5);
       if (j = (strcmp(ai,"array") == 0 && strlen(indata[i])
       5))
   strcpy(indata[i],"6");
```

```
         if (j = (strcmp(ai, "array") == 0 && strlen(indata[i])
     5)){
     indata[i][0] = '6';
     indata[i][1] = indata[i][5];
     indata[i][2] = '\0';
         }
         if (j != 0){
     io(11);
    return;
         }
         j = 1;
       }

       gets(input);
       transfer(input);
       if (strcmp(tran[0],"values") != 0){
   io(11);
   return;
       }
       j = 2;
       for (i= 2; i< clock+2 ; i++) {
    switch(indata[i][0]) {
    case '1': record = head;
       strcpy(names,tran[i-1]);
       break;
    case '2': strcpy(potmp1->name,names);
       for (u = strlen(names); u < 8; u++)
          strcat(potmp1->name," ");
       while((strcmp(potmp1->name, record->name) > 0)
      && (record->link != NULL)) {
      tp2 = record;
      record = record->link;
       }
       while((strcmp(potmp1->name, record->name) == 0)
     && (strcmp(tran[i-1], record->date) >= 0)
     && (record->link != NULL)) {
     tp2 = record;
     record = record->link;
       }
       strcpy(potmp1->date,tran[i-1]);
       strcpy(potmp1->spec,"       ");
       strcpy(potmp1->ph,"       ");
       strcpy(potmp1->hard,"        ");
       strcpy(potmp1->array,"        ");
       strcpy(potmp1->disch,"        ");
       strcpy(potmp1->temp,"        ");
       strcpy(potmp1->rect,"        ");
       break;
    case '3': strcpy(potmp1->spec,"");
       for (u = strlen(tran[i-1]); u<7; u++)
           strcat(potmp1->spec," ");
           strcat(potmp1->spec,tran[i-1]);
       break;
    case '4': strcpy(potmp1->ph,"");
       for (u = strlen(tran[i-1]); u<8; u++)
```

```
            strcat(potmp1->ph," ");
            strcat(potmp1->ph, tran[i-1]);
        break;
     case '5': strcpy(potmp1->hard,"");
        for (u = strlen(tran[i-1]); u <8; u++)
            strcat(potmp1->hard," ");
            strcat(potmp1->hard, tran[i-1]);
        break;
     case '6': strcpy(potmp1->array,"");
        if (indata[i][1] == ''){
           for (u = strlen(tran[i -1]);u < 8; u++)
      strcat(potmp1->array," ");
      strcat(potmp1->array, tran[i-1]);
        }
        else {
           strcpy(potmp1->array,"        ");
           potmp1->array[indata[i][1] - '0'] = tran[i-1][0];
        }
        break;
     case '7': strcpy(potmp1->disch,"");
        for (u =strlen(tran[i-1]); u< 8; u++)
          strcat(potmp1->disch," ");
          strcat(potmp1->disch, tran[i-1]);
        break;
     case '8': strcpy(potmp1->temp,"");
        for(u = strlen(tran[i-1]); u< 8; u++)
          strcat(potmp1->temp, " ");
          strcat(potmp1->temp, tran[i-1]);
        break;
     case '9': strcpy(potmp1->rect,"");
        for(u = strlen(tran[i-1]); u< 8; u++)
          strcat(potmp1->rect, " ");
          strcat(potmp1->rect, tran[i-1]);
        break;
     default : io(11);
        return;
     }
        }
        if (tp2 == NULL) {              //insert at the front first
        entry
    potmp1->link = head;
    head = potmp1;
        }
        else {
    potmp1->link = tp2->link;
    tp2->link = potmp1;
        }
        displ();
}


void river::delet()
{
      char m;
      char result[20];
```

```
  char set[10][30], names[20];
  int i, flag, z, flag1, u;


    i = 0;
    do {                          // read the input
gets(set[i++]);
m = set[i-1][strlen(set[i-1]) -1];
    } while (m != ';');

    transfer1(set[0]);
    if (strcmp(tran[0],"from") != 0) {
io(17);
return;
    }
    strcpy(indata[1],tran[1]);
    read_file();
    record = head;                // create a new river
    tp2 = NULL;                   // sets node point to empty
    if (record == NULL){
io(2);
return;
    }
    transfer1(set[1]);
    if (strcmp(tran[0],"where") != 0) {
io(18);
return;
    }
    for (i = 0; i<9; i++){
if (strcmp(tran[1],max[i]) == 0)
   flag = i;
    }
    z = 1;
    flag1 = 0;
    while( record != NULL) {
      switch(flag) {
      case 0 : strcpy(names, tran[2]);
 for (u = strlen(tran[2]); u < 8; u++)
   strcat(names," ");
   while ((record != NULL)
     && ( z = strcmp(record->name, names) != 0)){
     tp2 = record;
     record = record->link;}
 break;
      case 1 : while ((record != NULL)
   && ( z = strcmp(record->date, tran[2]) != 0)){
   tp2 = record;
   record = record->link;}
 break;
      case 2 : while((record != NULL)
   && ( z = (a2f(record->spec) - a2f(tran[2])) != 0)){
   tp2 = record;
   record = record->link;}
 break;
```

```
     case 3 : while((record != NULL)
  && ( z = (a2f(record->ph) - a2f(tran[2])) != 0)){
  tp2 = record;
  record = record->link;}
break;
     case 4 : while((record != NULL)
  && ( z = (a2f(record->hard) - a2f(tran[2])) != 0)){
  tp2 = record;
  record = record->link;}
break;
     case 5 : if (strlen(tran[2]) == 8) {
   while((record != NULL)
     && (z = strcmp(record->array,tran[2]) != 0)) {
     tp2 = record;
     record = record->link;}
   }
if(strlen(tran[3]) == 1)
while((record != NULL)
  && ( z = ((record->array[tran[2][0]-'0']-'0')
  - (tran[3][0] -'0')) != 0 )) {
  tp2 = record;
  record = record->link;
}
break;
     case 6 : while((record != NULL)
  && (z=(a2f(record->disch)-a2f(tran[2])) !=0)){
  tp2 = record;
  record = record->link; }
break;
     case 7 : while ((record != NULL)
  && (z=(a2f(record->temp) - a2f(tran[2])) != 0)){
  tp2 = record;
  record = record->link;}
break;
     case 8 : while((record != NULL)
  && ( z = strcmp(record->rect, tran[2]) != 0)){
  tp2 = record;
  record = record->link;}
 break;
    default : io(18);
return;
     }
     if (tp2 == NULL && z == 0){
flag1 = 1;
head = record->link;
record = record->link;
     }
     else if (z == 0) {
tp2->link = record->link;
record = tp2->link;
flag1 = 1;
     }                         // no record deleted
     else if (flag1 == 0 && record == NULL)
io(3);          // delete finished
```

```
        else
  io(4);
    }
   displ();                                   // display the data
}


void river::read_file()
{
     if (strcmp(indata[1], filenames) != 0) {
 head = new river;
 if ((in = fopen(indata[1],"r")) == NULL) {
   io(14);
   exit(0);
 }
     }
     record = head;                           // sets node point to
     empty
     while (fgets(input,85,in) != 0) { // read file
 mid(record->name,input,1,8);
 mid(record->date,input,9,10);
 mid(record->spec,input,19,7);
 mid(record->ph,input,26,8);
 mid(record->hard,input,34,8);
 mid(record->array,input,42,8);
 mid(record->disch,input,50,8);
 mid(record->temp,input,58,8);
 mid(record->rect,input,66,8);
 tail = new river;
 record->link = tail;
 record = tail;
 record->link = NULL;
     }
     strcpy(filenames, indata[1]);
}


void river::write_file()
{
     record = head;
     while (record->link != NULL) {
       strcpy(output,record->name);
       strcat(output,record->date);
       strcat(output,record->spec);
       strcat(output,record->ph);
       strcat(output,record->hard);
       strcat(output,record->array);
       strcat(output,record->disch);
       strcat(output,record->temp);
       strcat(output,record->rect);
       fprintf(in,"%s\n",output);
       record = record->link;
     }
}
```

```
void river::menu()
{
   char ans;

   do {
     io(6);
     gets(line);
     switch(toupper(line[0])) {
       case 'S' : selec();           // select
    break;
       case 'D' : delet();           // delete
    break;
       case 'I' : inser();           // insert
    break;
       case 'M' : modif();           // modify
    break;
       case 'Q' :  break;             // quit
       default  :  io(5);
     }
   } while (toupper(line[0]) != 'Q');
}


void river::read_input()
{
   char tr1[10];
   char input[50];
   int i;

     for (i = 0; i < 10; i++)
       strcpy(indata[i],"");

     clock = 0;
     gets(input);
     strcpy(indata[0],"");
     clock1 = clock;
     for (i = 0; i<strlen(input); i++) {
       if ((isdigit(input[i])) || (isalpha(input[i])) ||
(input[i] == '.')
   || (input[i] == '+') || (input[i] == '-') || (input[i] ==
   || (input[i] == '/')) {
   if (clock == clock1)
   ;
   else {
     clock++;
     clock1 = clock;
   }
   strcpy(tr1,"");
   tr1[0] = input[i];
   tr1[1] = '\0';
   strcat(indata[clock], tr1);
       }
       else if ( (input[i] == ' ') || (input[i] == '(') ||
       (input[i] == ')')
  || (input[i] == '=') || (input[i] == ',') || (input[i] ==
  || (input[i] == '"'))
```

```
    clock1++;
     }
}


void river::transfer(char input[50])
{
   char tr2[5];
   int i;

     for (i = 0 ; i < 10; i++)
       strcpy(tran[i],"");
     clock = 0;
     clock1 = clock;
     for (i = 0; i<strlen(input); i++) {
       if ((isdigit(input[i])) || (isalpha(input[i])) ||
(input[i] == '.')
  || (input[i] == '+') || (input[i] == '-') || (input[i] ==
  || (input[i] == '/')) {
  if (clock == clock1)
  ;
  else {
    clock++;
    clock1 = clock;
  }
  strcpy(tr2,"");
  tr2[0] = input[i];
  tr2[1] = '\0';
  strcat(tran[clock], tr2);
     }
     else if ( (input[i] == ' ') || (input[i] == '(') ||
     (input[i] == ')')
       || (input[i] == '=') || (input[i] == ',') || (input[i]
';')
       || (input[i] == '"'))
       clock1++;
   }
}


void river::transfer1(char input[50])
{
   char tr2[5];
   int i;

     for (i = 0 ; i < 10; i++)
       strcpy(tran[i],"");

     clock = 0;
     clock1 = clock;
     for (i = 0; i<strlen(input); i++) {
       if ((isdigit(input[i])) || (isalpha(input[i])) ||
(input[i] == '.')
  || (input[i] == '+') || (input[i] == '-') || (input[i] ==
  || (input[i] == '/')) {
    clock++;
    clock1 = clock;
  }
  strcpy(tr2,"");
  tr2[0] = input[i];
  tr2[1] = '\0';
  strcat(tran[clock], tr2);
       }
       else if ( (input[i] == ' ') || (input[i] == '(') ||
       (input[i] == ')')
  || (input[i] == '=') || (input[i] == ',') || (input[i] ==
  || (input[i] == '"') || (input[i] == '[') || (input[i] ==
  clock1++;
     }
}


// method modify()
void river::modif()
{
   char result[20];
   char set[10][30], updt[10][10][15], tran[10][20], tr1[4],
   names[20];
   int y, clock, i, p, flag, k, a, j, flag1, u, x,xx, clock1;
   double a2f(), z;

     strcpy(indata[1],"");
     for (i = 7; i <strlen(line); i++){
       tr1[0] = line[i];
       tr1[1] = '\0';
       strcat(indata[1],tr1);
     }
     read_file();
     i = 0;
     do {                              // read the input from screen
 mid(result, gets(set[i++]), 1, 5);
 y = strcmp(result, "where");
     } while (y != 0);

     for (k = 0; k < 10; k++)
       strcpy(tran[k],"");

     clock = 0;
     strcpy(tran[0],"");
     clock1 = clock;
     for (k = 0; k<strlen(set[i-1]); k++){
       if ((isdigit(set[i-1][k])) || (isalpha(set[i-1][k]))
       || (set[i-1][k] == '.') || (set[i-1][k] == '+') ||
       1][k] == '-')
       || (set[i-1][k] == '*') || (set[i-1][k] == '/')) {
  if (clock == clock1)
    ;
  else {
    clock++;
    clock1 = clock;
  }
  strcpy(tr1,"");
  tr1[0] = set[i-1][k];
  tr1[1] = '\0';
  strcat(tran[clock], tr1);
       }
       else if ((set[i-1][k] == ' ') || (set[i-1][k] == '(')
  ||(set[i-1][k] == ')') ||(set[i-1][k] == '=')|| (set[i-1][k]
',')
  ||(set[i-1][k] == ';') ||(set[i-1][k] == '[')|| (set[i-1][k]
']'))
  clock1++;
     }
     if (strcmp(tran[0],"where") != 0) {
       io(17);
       return;
     }

     for (p = 0; p<9; p++){
       if (strcmp(tran[1],max[p]) == 0)
  flag = p;
     }

     for(i = 0; i<10; i++)
       for (j = 0; j<10; j++)
  strcpy(updt[i][j],"");

     record = head;
     if (record == NULL){
       io(2);
       return;
     }
     z = 1;
     while( record != NULL) {
     switch(flag) {
      case 0 : strcpy(names, tran[2]);
        for (u = strlen(tran[2]); u < 8; u++)
   strcat(names," ");
        while ((record != NULL)
   && (z = strcmp(record->name, names) != 0)){
   record = record->link;}
        break;
      case 1 : while ((record != NULL)
   && (z = strcmp(record->date, tran[2]) != 0)){
   record = record->link;}
        break;
      case 2 : while ((record != NULL)
   && (z = (water::a2f(record->spec)
   - water::a2f(tran[2])) != 0)) {
   record = record->link;}
        break;
      case 3 : while ((record != NULL)
   && (z = (water::a2f(record->ph)
   - water::a2f(tran[2])) != 0)){
   record = record->link;}
        break;
      case 4 : while ((record != NULL)
   - water::a2f(tran[2])) != 0)){
   record  = record->link;}
        break;
      case 5 : if (strlen(tran[3]) != 1)
   while ((record != NULL)
     && (z = strcmp(record->array,tran[2]) != 0)){
     record = record->link;
   }
        if (strlen(tran[3]) == 1)
   while ((record != NULL)
     && (z = ((record->array[tran[2][0]-'0']-'0')
     - (tran[3][0] - '0')) != 0)){
     record = record->link;
   }
        break;
      case 6 : while ((record != NULL)
   && ( z =(water::a2f(record->disch)
   - water::a2f(tran[2])) != 0)){
   record = record->link;
  }
        break;
      case 7 : while ((record != NULL)
   && (z =(water::a2f(record->temp)
   - water::a2f(tran[2])) != 0)){
   record = record->link;
        }
        break;
      case 8 : while ((record != NULL)
   && ( z = strcmp(record->rect, tran[2]) != 0)){
   record = record->link;}
        break;
      default : io(17);
  return;
      }

     if (z==0){
       flag1 = 1;
     for (x = 0; x< 10; x++)
       for (xx = 0; xx < 10; xx++)
  strcpy(updt[x][xx],"");

     for(k = 0; k < i-1; k++){
       clock = 0;
       clock1 = clock;
       for (j = 0; j < strlen(set[k]); j++){
  if((isdigit(set[k][j])) || (isalpha(set[k][j])) || (set[k]
== '.')
    || (set[k][j] == '+') || (set[k][j] == '-') || (set[k][j]
'*')
    || (set[k][j] == '/'))  {
    if (clock == clock1)
      ;
    else {
```

```
    strcpy(tr1,"");
    tr1[0] = set[k][j];
    tr1[1] = '\0';
    strcat(updt[k][clock], tr1);
  }
  else if((set[k][j] == ' ') || (set[k][j] == '(')
    || (set[k][j] == ')') || (set[k][j] == '=') || (set[k][j]
',')
    || (set[k][j] == ';') || (set[k][j] == '[') || (set[k][j]
']'))
      clock1++;
      }

      if (k == 0){
 if (strcmp(updt[0][0],"set") != 0){
   io(17);
   return;
 }
 else {
   for (p = 0; p<clock ; p++)
     strcpy(updt[0][p],updt[0][p+1]);
 }
      }
   }

   for (y = 0; y< i-1; y++) {
     for (p = 0; p <9; p++) {
       if (strcmp(updt[y][0],max[p]) == 0)
       switch(p) {
 case 0 : strcpy(record->name,updt[y][1]);
   for (u =strlen(updt[y][1]); u <8; u++)
      strcat(record->name," ");
   break;
 case 1 : strcpy(record->date, updt[y][1]);
   break;
 case 2 : if(updt[y][2][0] != '' && (strcmp(updt[y][1],"spec")
0))
     strcpy(updt[y][1],
     gap(updt[y][2][0], record->spec, updt[y][3]));
   strcpy(record->spec,"");
   for ( u = strlen(updt[y][1]); u <7 ; u++)
     strcat(record->spec," ");
     strcat(record->spec,updt[y][1]);
   break;
 case 3 : if (updt[y][2][0] != '' && (strcmp(updt[y][1],"ph")
0))
     strcpy(updt[y][1],
     gap(updt[y][2][0], record->ph, updt[y][3]));
   strcpy(record->ph,"");
   for ( u = strlen(updt[y][1]); u <8 ; u++)
     strcat(record->ph," ");
     strcpy(updt[y][1],
     gap(updt[y][2][0], record->hard, updt[y][3]));
   strcpy(record->hard,"");
   for (u = strlen(updt[y][1]); u <8 ; u++)
```

```
      strcat(record->hard," ");
      strcat(record->hard,updt[y][1]);
    break;
  case 5 : if (strlen(updt[y][1]) == 8)
      strcpy(record->array,updt[y][1]);
    else if((strlen(updt[y][2])== 1)
      && (strcmp(updt[y][0],"array") ==0))
    record->array[updt[y][1][0] - '0'] = updt[y][2][0];
    else if ((strlen(updt[y][2])==5)
      && (strcmp(updt[y][2],"array")==0)){
      updt[y][2][0] = gap1(updt[y][4][0],
      record->array[updt[y][3][0]-'0'],updt[y][5][0]);
      record->array[updt[y][1][0] - '0'] = updt[y][2][0];
    }
    break;
  case 6 : if(updt[y][2][0] != '' && (strcmp(updt[y][1],"dis
== 0))
      strcpy(updt[y][1],
      gap(updt[y][2][0], record->disch, updt[y][3]));
    strcpy(record->disch,"");
    for ( u = strlen(updt[y][1]); u <8 ; u++)
      strcat(record->disch," ");
      strcat(record->disch,updt[y][1]);
    break;
  case 7 : if (updt[y][2][0] != '' && (strcmp(updt[y][1],"tem
== 0))
      strcpy(updt[y][1],
      gap(updt[y][2][0], record->temp, updt[y][3]));
    strcpy(record->temp,"");
    for (u = strlen(updt[y][1]); u <8 ; u++)
      strcat(record->temp," ");
      strcat(record->temp,updt[y][1]);
    break;
  case 8 : strcpy(record->rect,"");
    for (u =strlen(updt[y][1]); u <8; u++)
      strcat(record->rect," ");
    strcat(record->rect, updt[y][1]);
    break;
        }
      }
    }
    if (record != NULL)
      record = record->link;
   }
  }
   if (flag1 != 1)

}


// method initial()
void river::init()
{
     strcpy(max[0],"name");
     strcpy(max[1],"date");
     strcpy(max[2],"spec");
     strcpy(max[3],"ph");
     strcpy(max[4],"hard");
     strcpy(max[5],"array");
     strcpy(max[6],"disch");
     strcpy(max[7],"temp");
     strcpy(max[8],"place");
     strcpy(max[9],"diag");
     strcpy(max[10],"area");
     strcpy(max[11],"circ");
}


// method select()
void river::selec()
{
   char m;
   char result[20], value[10];
   char set[10][30], tran1[10][20], tr1[4], names[20];
   int i, flag, flag1, j, k, sign, flag2, u;
   double z;
   double a2f();


     i = 0;
     do {                                    // read the input
     screen
       gets(set[i++]);
       m = set[i-1][strlen(set[i-1]) -1];
     } while (m != ';');

     transfer1(set[0]);
     if (strcmp(tran[0],"from") != 0) {
       io (17);
       return;
     }
     strcpy(indata[1], tran[1]);
     read_file();
     if ((strcmp(set[1],"")) != 0) {
       transfer1(set[1]);
       if (strcmp(tran[0],"where") != 0){
  io(19);
       return;
       }
       flag2 = 10;
       if (strcmp(tran[3],"and") == 0)
  for (i = 0; i <9; i++) {
    if (strcmp(tran[4], max[i]) == 0)
      flag2 = i;
  }
       for (i = 0; i<9; i++){
  if (strcmp(tran[1],max[i]) == 0)
    flag = i;
```

```
        }

      record = head;
      tp2 = NULL;
      if (record == NULL){                // empty file
        io(2);
        return;
      }

    while( record != NULL) {
    z = 1;
    switch(flag) {
     case 0 : strcpy(names,tran[2]);
       for (u = strlen(tran[2]); u <8; u++)
         strcat(names," ");
       while ((record != NULL)
         && (z = strcmp(record->name,names) != 0)){
         record = record->link;}
       break;
    case 1 :  while ((record != NULL)
         && (z = strcmp(record->date, tran[2]) != 0)){
         record = record->link;}
       break;
    case 2 :  while ((record != NULL)
         && (z = (water::a2f(record->spec)
         - water::a2f(tran[2])) != 0)){
         record = record->link;}
       break;
    case 3 :  while ((record != NULL)
         && (z = (water::a2f(record->ph) - water::a2f(tran[2]))
0)){
         record = record->link;}
       break;
    case 4 :  while ((record != NULL)
   && (z = (water::a2f(record->hard)
   - water::a2f(tran[2])) != 0)){
   record = record->link;}
       break;
    case 5 :  while ((record != NULL)
         && (z = ((record->array[tran[2][0]-'0']-'0')
         - (tran[3][0]- '0')) != 0)){
         record = record->link;}
       break;
    case 6 :  while ((record != NULL)
         && (z = (water::a2f(record->disch)
         - water::a2f(tran[2])) != 0)){
         record = record->link;}
       break;
    case 7 :  while ((record != NULL)
   && (z = (water::a2f(record->temp)
   - water::a2f(tran[2])) != 0)){
   record = record->link;}
       break;
    case 8 :  while ((record != NULL)
         && (z = strcmp(record->rect, tran[2]) != 0)){
```

```
       record = record->link;}
     break;
  default : io(19);
     return;
 }

 if (strcmp(tran[3],"and") == 0) {
    switch(flag2) {
    case 0 : strcpy(names, tran[5]);
      for (u = strlen(tran[5]); u < 8; u++)
 strcat(names," ");
      z = strcmp(record->name, names);
      break;
    case 1 : z = strcmp(record->date, tran[5]);
      break;
    case 2 : z = (water::a2f(record->spec) -
water::a2f(tran[5]));
      break;
    case 3 : z = (water::a2f(record->ph) - water::a2f(tran
      break;
    case 4 : z = (water::a2f(record->hard) -
water::a2f(tran[5]));
      break;
    case 5 : z =((record->array[tran[5][0]- '0'] - '0') -
(tran[6][0] - '0'));
      break;
    case 6 : z = (water::a2f(record->disch) -
water::a2f(tran[5]));
      break;
    case 7 : z = (water::a2f(record->temp) -
water::a2f(tran[5]));
      break;
    case 8 : z = strcmp(record->rect, tran[5]);
      break;
    default: io(19);
      return;
   }
 }


 clock = 0;
 strcpy(tran1[0],"");
 for (i = 0; i<strlen(line); i++){
   switch(line[i]) {
    case ' ' : clock++;
 strcpy(tran1[clock],"");
 break;
    case ',' : break;
    case '.' : break;
    case '(' : break;
    case ')' : break;
    default  : strcpy(tr1, "");
 tr1[0] = line[i];
 tr1[1] = '\0';
 strcat(tran1[clock], tr1);
 break;
   }
 }
```

```
  if (strcmp(tran1[0],"select") != 0){
    io(19);
    return;
  }

// select *, get full details of all properties
  if (strcmp(tran1[1],"*") == 0){
    strcpy(tran1[1],"name");
    strcpy(tran1[2],"date");
    strcpy(tran1[3],"spec");
    strcpy(tran1[4],"ph");
    strcpy(tran1[5],"hard");
    strcpy(tran1[6],"array");
    strcpy(tran1[7],"disch");
    strcpy(tran1[8],"temp");
    strcpy(tran1[9],"place");
    clock = 9;
  }

  if (z == 0){
     for (j = 0; j <= clock; j++){
       sign = 0;
     for (i = 0; i< 12; i++) {
      if (strcmp(tran1[j],max[i]) == 0)
      switch(i) {
       case 0 : printf("%s ",record->name);
  break;
       case 1 : printf("%10s ",record->date);
  break;
       case 2 : printf("%s ",record->spec);
  break;
       case 3 : printf("%s ",record->ph);
  break;
       case 4 : printf("%s ",record->hard);
  break;
       case 5 : printf("%s ",record->array);
  break;
       case 6 : printf("%s ",record->disch);
  break;
       case 7 : printf("%s ",record->temp);
  break;
       case 8 : printf("%s", record->rect);
  break;
       case 9 : if (strcmp(tran1[j + 1], "Place") == 0)
  j++;
  printf(" %d ",diagonal(record->rect));
  break;
       case 10: if (strcmp(tran1[j + 1], "Place") == 0)
  j++;
  printf(" %d ",area(record->rect));
  break;
       case 11: if (strcmp(tran1[j + 1], "Place") == 0)
  j++;
  printf(" %d ",circum(record->rect));
```

```
 break;
      }
     mid(value, tranl[j],1,5);
     if (strcmp(value,"array") == 0 && (strlen(tranl[j]) >
     sign == 0){
mid(value, tranl[j],7,1);
k = value[0] - '0';
printf(" %c ", record->array[k]);
sign = 1;
     }
    }
   }
   printf("\n");
  }
  if (record != NULL)
   record = record->link;
 }
}
 else
  displ();
  printf("\n");
}


// method input/output()
void river::io(int i)
{
  switch(i) {
   case 1: puts("NAME    |DATE     |SPEC   |PH     |HARD
   |ARRAY   |DISCH   |TEMP    |PLACE");
    break;
   case 2: puts(" No data in this file");
    break;
   case 3: puts("No record deleted");
    break;
   case 4: puts("Delete finished");
    break;
   case 5: puts("");
    puts(" Not correct !! Please Try Again!");
    break;
   case 6: puts("Please enter your choice (Q to quit)!");
    break;
   case 7: puts("");
    puts("Save data file ? (Y or N)");
    puts("");
    break;
   case 8: puts("Now saving data file...");
    puts("");
    break;
   case 9: puts("Data File do not save !!");
          io(13);
    break;
  case 10: puts("");
    puts("*****  If you want to continueous please enter <c>
    break;
  case 11: puts("You have a wrong insert format");
    break;
  case 12: puts("Modify finished");
    break;
  case 13: puts("");
    io(21);
    puts("");
    break;
  case 14: puts("Can't open the file");
    break;
  case 15: puts("");
    puts("Sure ? (Y or N)");
    break;
  case 16: puts("No record modified");
    break;
  case 17: puts("You have a wrong update format");
    break;
  case 18: puts("You have a wrong delete format");
    break;
  case 19: puts("Your have a wrong select format");
    break;
  case 20: strcpy(output,record->name);
    strcat(output,record->date);
    strcat(output,record->spec);
    strcat(output,record->ph);
    strcat(output,record->hard);
    strcat(output,record->array);
    strcat(output,record->disch);
    strcat(output,record->temp);
    strcat(output,record->rect);
    puts(output);
    break;
  case 21: puts("=======|=========|======|=======|========|
   =======|=======|=======|========");
    break;
  }
}


// method display()
void river::displ()
{
   char test1[2];
   int i = 0;
     record = head;
     test1[0] = 'C';
     while (test1[0] == 'C')  {
       io(21);
       io(1);
       io(21);
       while ((record->link != NULL) && (i < 15)) {
  io(20);
  record = record->link;
  i++;
       }
       test1[0] = ' ';
       if (record->link != NULL){
 io(10);
 gets(test1);
 test1[0] = toupper(test1[0]);
 i = 0;
       }
     }
     return;
}


char* river::mid(char *ds, char *ss, int a, int b)
{
   int k, n;

     n = strlen(ss);
     if (( 0< a && a <= n) && (0<b && b <= n) && (a + b -1 <
     {
       for (k = a; k <a +b; k++)
  ds[k-a] = ss[k -1];
  ds[b] = '\0';
     }
     else
       ds[0] = '\0';
     return ds;
}


// Main object oriented data model simulation parogram
// The main program that uses the classes and subclasses of
// hydraulic database system
main()
{
   // object declarations
   // station s;
   river r;
   char key1[2];

     r.init();
     r.menu();
     puts("Save data file ? ( Y or N) ");
     gets(key1);
     if (toupper(key1[0]) == 'Y') {
       puts("Now Saving Data File......");
       if (freopen("p","w",in) == NULL) {
  puts("can't write into datafile !! ");
  exit(1);
       }
       r.write_file();
       puts("Saving Complete !!");
       fclose(in);
     }
```

```
    else
      puts("Data File not save !!");
}
```

VITA

Hui-Chen Nee

Candidate for the Degree of

Master of Science

Thesis: INTEGRATING AN OBJECT ORIENTED PROGRAMMING LANGUAGE SYSTEM WITH A DATABASE SYSTEM

Major Field: Computer Science

Biographical:

Personal Data: Born in Taipei, Taiwan, R. O. C., May 1, 1963, the daughter of Chinghui Nee and Shunu Chou.

Education: Graduate from KingMei High School, Taipei, Taiwan, in May 1982; received Bachelor of Engineering Degree from Tamkang University in May, 1987; completed the requirements for the Master of Science degree at Oklahoma State University in May, 1991.

Professional Experience: Programmer, Department of Civil Engineering, National Taiwan University, May, 1987, to May, 1988.