9-12-1988

ClassC/Elaine: a multiple inheritance object oriented C language

Paul Kirkaas

Rochester Institute of Technology

School of Computer Science and Applied Technology

ClassC/Elaine:

A Multiple Inheritance Object Oriented C Language

by

Paul Kirkaas

A thesis submitted to the faculty of the School of Computer Science and Technology in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Approved by:

James Heliotis 9/13/88

Dr James Heliotis

Andrew Kitchen 13 Sept 88

Dr Andrew Kitchen

Peter G. Anderson 13 Sept 88

Dr Peter Anderson

ClassC/Elaine:

A Multiple Inheritance Object Oriented C Language

by Paul Kirkaas

Paul Kirkaas

Paul Kirkaas

12 September 1988

ABSTRACT

Object oriented programming is a way of abstracting information and operations to make programming more efficient and reliable. *C* is a non object oriented programming language that has become a de facto language standard in academic and industrial applications because of its power and flexibility. ClassC is an attempt to add object orientation on the existing framework that C provides. Specifically, ClassC adds the new data type *object* , and the new aggregate declaration, *class*.

ClassC differs from other C-based object oriented languages such as C++ Objective-C in three points:

1) ClassC provides true multiple inheritance, which is very commonly discussed in the description of object oriented languages, but very rarely actually implemented.

2) ClassC provides both strict type checking on objects as well as typeless dynamic binding – a variable of type *object* may be assigned any class instantiation. Dynamic binding is typical of interpreted languages like lisp and Smalltalk, but rare in compiled languages like *C* and *Pascal,* etc.

3) ClassC offers automatic garbage collection of memory no longer referenced by any object variables.

ClassC/Elaine is intended to be a flexible system that allows programmers to explore different qualities of object oriented programming, including strong and weak type checking, and single and multiple inheritance. This thesis is a description of the background, design, and implementation of the ClassC language.

Keywords

Object Oriented Programming, C++, Objective-C, Smalltalk, ClassC

Computing Review Subject Codes

Primary Code: D.3.2 Language Classifications

Secondary Code: D.3.3 Language Constructs

CONTENTS

1. Introduction

Object oriented programming is a way of abstracting information and operations to make programming more efficient and reliable. *C* is a non object oriented programming language that has become a de facto language standard in academic and industrial applications because of its power and flexibility. ClassC is an attempt to add object orientation on the existing framework that C provides.

1.1 ClassC/Elaine

ClassC is a C-based object oriented programming language. The language adds object oriented capabilities to the C programming language. Specifically, ClassC adds the new data type *object*, and the new aggregate declaration, *class*. Class definitions are like C structures except they may contain function declarations, and they may inherit components from multiple ancestors; objects are instances of a particular class. An object variable may be declared by a class name, in which case it must represent only objects of that class or descendants of that class, or an object variable may be declared by the keyword *object*, in which case it can be used to represent any type of object in the system with no type checking.

Elaine is the programming environment for ClassC. Initially it consists of the runtime library functions necessary to implement ClassC; enhancements will be added over time.

There are other C-based object oriented languages, such as C++ and Objective-C, but ClassC differs from these other languages in three points.

1) ClassC provides true multiple inheritance, which is very commonly discussed in the description of object oriented languages, but very rarely actually implemented.

2) ClassC provides both strict type checking on objects as well as typeless dynamic binding -- a variable of type *object* may be assigned any class instantiation. Dynamic binding is typical of interpreted languages like Lisp and Smalltalk, but rare in compiled languages like *C* and *Pascal*.

3) ClassC offers automatic garbage collection of memory no longer referenced by any object variables.

1.2 Motivation

Object oriented programming provides a way to abstract data and control. It allows the programmer to create models that are internally complex and sophisticated, yet which present a simple and uniform interface to the user. There are two broad arguments for Object Oriented programming.

The first is that Object Oriented programming improves productivity and reliability by allowing the reuse of code and the standardization of common data types and operations.

The second argument is that by allowing greater abstraction, object oriented languages are continuing the trend in programming languages towards a more human like and less machine like means of expressing concepts. This implies that object oriented programming can facilitate not only bigger programs, but cleverer ones as well. This is my primary interest.

The C language was designed with a particular philosophy as well; and one which often conflicts with many of the common arguments for object oriented programming. In general, C is designed for efficiency and enforces fairly casual type checking. C allows the programmer unusual intimacy with the data representation and the machine. There is no information hiding with C.

These characteristics give the programmer flexibility and the ability to coerce the language in unusual ways. This is valuable, but can also be dangerous for unsophisticated users.

Object oriented languages emphasize much more information hiding and abstraction. The user is isolated from the machine and data representation as much as possible.

Both these philosophies have merit. The task of combining these features into a single language is to some degree a subjective one, and one which always makes some compromise for every decision made.

For example, in most object oriented languages, the data components can not be accessed directly. The user can only change the data component through a member function (or "message"). This preserves abstraction, but is less efficient, more time consuming to write, and irritating. The choice made for ClassC was to allow the programmer direct access to data components of objects.

The goal of ClassC/Elaine, then, is to provide a usable C based object oriented language that offers an alternative to previous implementations. Most of the code executed in any program is straight forward and does not benefit from an object oriented approach (e.g, for-loops, arithmetic operations, etc). This code should be implemented as efficiently as possible for rapid execution. This is the motivation for using C as the base language. Object type data manipulations should be used for operations which are conceptually intricate, where use of object oriented techniques could give both the programmer and the program more clarity and expressiveness. This is the motivation for enhancing C with object oriented features; moreover, this is the reason for giving ClassC a higher level of abstraction that C++ or ObjectiveC through multiple inheritance and untyped objects.

In order to be genuinely useful, ClassC was made compatible with existing C language facilities. It is compatible with all existing standard #include files, as well as all library packages. It allows separate compilation and linking of ClassC modules, as well as the creation of individual ClassC libraries.

1.3 Background

The foundations of Object Oriented programming were laid by the programming language Simula, which was developed in Norway in the early 1960's [DAH66]. It introduced the concept of user defined classes, and was developed as a simulation language for modeling complex systems. The power of this concept was not widely recognized until Alan Kay and the research group at Xerox PARC developed the Smalltalk interactive environment. It evolved from Dr. Kay's concept of a computer called the Dynabook, which would be very user friendly and suitable as a learning aid for

children [GOL83]. Smalltalk remains to this day the prototypical example of an object oriented language. It is a rich and complex environment that shows the power of the object oriented concept.

Yet this richness and complexity has a price. Smalltalk is big, slow, expensive, and available on very few machines. Its primary drawback from a systems development perspective is its slowness. Smalltalk runs on a virtual machine / interpreter, which is inappropriate for large software systems. This is unfortunate, because software systems are classic candidates for the productivity benefits of object oriented programming. Concepts like queues, stacks, hash tables, etc. are ideal abstraction types. But an operating system written in Smalltalk running on conventional hardware would be very slow.

Enter C++ [STR86]. C++ was developed at AT&T Bell Laboratories, home of Unix, C, Kernighan and Ritchie. It was developed by Bjarne Stroustrup with the intent of giving as much object oriented structure to C as possible, while sacrificing none of the efficiency of pure C code. It is therefore perhaps the best system development language available today.

The C++ compiler translates C++ code into standard C. Thus, C++ must implement its object oriented features in a C format. It does this by using standard C structures. An object in C++ is implemented in two ways. The function components of the class are given unique names and written to the output file. The function members are associated with the class rather than with the object. This means that all instances of a class have unique private data but share the same component functions.

The data portion of a C++ object is implemented in a standard C structure, with all the data components of the object as elements of the generated structure. Inheritance is implemented by copying the parent object structure and concatenating the new elements of the derived class after the elements of the parent class. The component sequence of the parent and derived classes are therefore identical up to the end of the parent class. Consequently, the address of an instance of a derived class can be assigned to an object pointer of a parent class and the standard C structure dereferencing operations for the parent will correctly dereference the derived object components.

This makes object dereference easy and efficient, but makes multiple inheritance difficult for reasons discussed later.

Objective-C is a language developed by Dr. Brad Cox at Productivity Products International and seems much like C++ with a different syntactic structure. The two languages offer similar features [COX86].

1.4 System Description

The ClassC/Elaine compiler was be developed under Unix using the Unix lexical analyzer **Lex** and the Unix parser generator **Yacc**, and uses the standard Unix/C libraries. The running system consists of three files-

1. The parser *efront*-

   The ClassC/Elaine front end parser, written in C. The parser translates ClassC/Elaine .e source files into ".c" C language files.

2. The Unix Shellscript *ce*-

   The shellscript *ce* is the compiler command invoked by the user. It parses the command line options and source files, applies the appropriate version of the parser and runtime library, and calls the C compiler *cc* on the result.

3. The ClassC/Elaine runtime routines, *libce.a*.

   libce.a contains runtime functions such as _new(), _deref(), etc.

1.5 Language Description

The concept for ClassC differs from those of other C-based object oriented languages. ClassC was developed as a learning tool for data abstraction and object oriented programming concepts.

Consequently, it emphasizes object oriented functionality at the sacrifice of some efficiency. This

philosophy is reflected in the decision to provide multiple inheritance and dynamic binding of objects. There is some runtime cost to provide these features, mainly function call overhead.

ClassC is a superset of C; that is, all C operations are supported by ClassC, and pre-existing library and object files as well as standard header files are all compatible with ClassC. The language was developed in a Unix operating system environment, using the Unix language development tools. The parser accepts standard C code and passes it to the output file unchanged. It maintains symbol tables and scoping information while building the parse tree.

In addition to standard C, the parser accepts and processes the following:

1. The aggregate definition *class*-

   A class declaration is much like a standard C structure declaration. A class is composed of *Members;* of which there are two types:

   a. *Variable* members-

      which are passive data components of the class instances.

   b. *Function* members-

      which are functions that act on the private member variables of class instances.

   In addition, a class definition may contain a list of parent classes from which components are inherited.

2. The class member function definition-

   Different classes may have member functions with the same name; e.g., *print*. To distinguish them, class member functions are defined using the class name and a scoping operator.

3. Objects-

   All objects are dynamically allocated at runtime by explicit calls to a *'new()'* function. A variable of type *object* is really an index into an object table. An object may be declared in one of two ways:

1. With a class name-

   Class names are used to declare object variables. Object variables declared this way can only represent objects of that particular class type or its descendants.

2. With the type specifier *object*-

   A variable declared with the type specifier *object* can be used to represent any type of object instance at all.

4. The object member dereference-

   The member dereferencing operator used in ClassC is ":>". C++ uses the ordinary C structure dereferencing operators for object dereferencing, but the dereferencing mechanism in ClassC is implemented differently and suggests the use of a separate symbol.

5. Variable declarations-

   Variable declarations can be made anywhere in a block. This is intended to more closely associate the variable declaration with its use. Every variable must still be declared *before* its first use.

1.6 Functional Overview

Classes:

A class is declared as follows:

```
class Class_name : SuperClass_1 : SuperClass_2 : ...
        {       type declarator;
                type declarator;
                type declarator;
                type declarator;
                        .
                        .
                        .
        };
```

where *SuperClasses* are optional and separated by the inheritance operator ":"; and where *declarator* is any type of legal ClassC declarator, including a function or a class object.

Two special types of functions that may be declared as a class member are *init()* and *delete()*. *init()* is that class's initializer function. The init() function is defined by the user and is automatically invoked when an object of that class is created. Otherwise, if init() is not declared, uninitialized memory space will be allocated.

delete() is the class delete function. It should be defined by the user to clean up any special memory resources which objects of that class may allocate to themselves (through *malloc()*, for instance).

Objects:

Objects are created dynamically from the heap; there are no static objects. Each object is an instance of a specific class. A member of an object (the member can be either variable or function) is dereferenced by the object dereferencing operator ":>". The general form of the object dereference is:

"*objectExpression* :> *memberName*;";

where "objectExpression" is usually just a variable name. In general, however, an object expression can also be the result of a pointer dereference, a return value from a function, etc. The type of the entity returned by the object dereferencing operator is determined by the type of the object expression. If the class to which the object expression belongs can be determined at compile time, the type of the returned dereferenced element is determined by its type as declared in the class definition.

If, however, the object expression evaluates to an instance of the generic class *object*, its true class cannot be determined at compile time. In this case the type of the result of the dereference is defaults to *object* as well. The programmer can cast the result to be of any type and override the default.

Member Functions:

Member functions are defined with the class name, scoping operator "::", and the function name -- e.g, "Son::print()". Arguments are declared just like in an ordinary C function definition. An object member function can reference the other member variables or functions of that same object

with no special dereferencing or casting required. A reference to "self" is passed automatically on a call to a member function, and may be used explicitly by the programmer in an object's member function definition to reference the object itself.

Examples:

Declare class **Son** to be a descendant of classes **Papa** and **Mama**.

```
class Son : Papa : Mama
        { int a;
          char * b;
          init(); /* Optional initialization function */
          void print();
        };
```

Define member function init() of class Son:

```
Son::init(x,y)     int x; char * y;
{ b = (char *) malloc(strlen(y) + 1);
  strcpy (b,y);
  a = x;
}
```

Declare an object & initialize to be of class Son:

```
Son xmpl = Son:>new(5,"Hello");
```

Send the message "print" to xmpl:

```
xmpl:>print();
```

Reset member element "b" of xmpl:

```
xmpl:>b = "Goodbye"
```

2. Functional Specification -- ClassC/Elaine Programming Guide

2.1 Introduction to ClassC/Elaine

ClassC is an object oriented programming language based on *C,* and it is assumed the user has a both a solid knowledge of the C language and a conceptual understanding of object oriented programming.

Like C++ and ObjectiveC, ClassC is really a preprocessor, converting ClassC source code into C, then into assembly language, and finally into machine code. ClassC is a superset of the C language, and all C constructs except *enum* are accepted and semantically unchanged (*enum* was not implemented because it differs from other aggregate types. It would require a special symbol table, and is not widely used in C. It would be a tedious but not difficult feature to add).

ClassC/Elaine has the following features:

ClassC's object oriented extensions to C

1. Class Definition: A class is defined like a C structure, except that it may inherit components from an arbitrary list of ancestor classes, and may include functions as components.

2. Objects: An instance of a class as defined above is an *object.* Objects can be declared in one of two ways. A class definition is like a typedef in C. This means that a variable can be declared by using a class name as a type declaration. Variables so declared can only be assigned objects of that same type or ancestor types. In addition, a new data type *object* is introduced, which can be used to declare variables which represent *any* class type.

3. Garbage Collection Option:

   Invoking the "ce" compile command with the option "-G" results in the inclusion of the ClassC garbage collection mechanism.

ClassC's non-object oriented extensions to C

1. Inline Comments:

Following C++, any text between "//" and the newline is ignored.

2. Statement/Declaration Mixing:

   Statements and variable declarations can be intermixed in the body of a block, as long as each variable is declared before its use.

The distinction between classes, objects, and object variables in ClassC should be understood. A class type definition is used to define the valid data components and operations (methods, messages) for objects of that class. An object is an instance of a particular class, created dynamically from the heap at runtime by explicit calls to "*className* :> new()". An object *variable* is implemented as an integer index into an object table. Since the type (class) of each object is kept in its object table entry, it need not be a characteristic of the object variable itself. This allows dynamic binding of object variables to objects.

In other C based object oriented languages, the type of the object variable must be declared to be of a specific class, and can be used represent only objects of that particular class, or classes derived from that class. In ClassC, type checking is performed only on those objects declared to be of a particular class. The declaration *object* can be used as a catch-all to hold any system object type with no compatibility checking. This is useful for collection classes such as sets which should be able to contain all object types.

2.2 Overview

This section introduces the flavor of programming in ClassC. More detailed explanations can be found in the following sections.

Objects in ClassC are much like structures which can have function components. The ":>" operator is used to dereference a component of an object. If the component is a data/variable type component, it can be used as an l-value -- that is, it can appear on the left side of an assignment expression. There is no concept of "private" components in ClassC.

A function component in ClassC is like a message in Smalltalk. Messaging could have been implemented with a Smalltalk like syntax, but one goal of ClassC was to be as C-like as possible while providing object oriented extensions. Hence, messages arguments are passed like regular function arguments in C. Thus the expression:

"thisObj :> message(arg1,arg2);"

applies the member function "message()" to the object "thisObj" with the arguments "arg1" and "arg2".

In ClassC, classes are defined by the inheritance of components from parent classes and an additional declaration section for components unique to the class being defined.

A class name may be used to declare an object variable, which then represents a pointer to an object of that class type. However, the object itself is not created until an explicit initialization call is made.

For example, the declaration

"MyClass objVariable;"

declares "objVariable" to be a pointer to an object of type "MyClass"; but it is initialized to the NULL pointer (0). To create a new object of type "MyClass", send the message "new()" to "MyClass" –

"MyClass:>new()".

It is possible to declare an object variable and initialize the object in one step:

"MyClass objVariable = MyClass:>new();"

Class names are also used when defining member functions. Since several classes may have member functions of the same name, the class name is prepended to the member function name to uniquely

identify it.

2.3 Defining classes

A class in ClassC is defined much like a structure in C. A class definition is indicated by the keyword "class", followed by the name of the class to be defined, an optional inheritance list of previously defined classes, and the body of the class definition. The definition body is just a sequence of declarations in a standard C format. This declaration set may include function and object declarations.

The form of a class definition is thus:

```
class Class_name : SuperClass_1 : SuperClass_2 : ...
        {       type declarator;
                type declarator;
                type declarator;
                type declarator;
                        .
                        .
                        .
        };
```

Three things to bear in mind about classes –

1. Class Definitions-

   No other classes, structures or typedefs may be *defined* within a class definition, although previously defined typedefs and structures may be used within a definition, and other classes which are defined elsewhere within the file (earlier or later) may be used to declare member components.

2. Class Names-

   The "class" keyword is used only on definition. Thereafter the defined class name is used like a new type name. This means that unlike structures, you cannot have a variable with the

same name as a class.

3. No Unnamed Classes-

   Unlike *struct* in C, all classes must be given a name; furthermore, it is impossible to declare an object instance of a class in the same statement as the class definition.

Class Inheritance

Classes may inherit components (data and methods) from an arbitrary number of antecedents. This implies a potential conflict when the same component name is used in more than one ancestor. This conflict is resolved as follows:

1. If the conflict involves two components of the same name but of different declared type in two or more different ancestors, this is declared to be a semantic error, compilation is aborted and an appropriate error message is reported.

2. If the conflict involves components of the same name and the same declared type in each ancestor, then the component from the parent earliest in the list is defaulted.

   This default inheritance of the component is irrelevant for data components, but critical for function components. That is, if the same function name and type is declared in several ancestors, the first definition encountered when traversing the list of ancestors is the function definition used for the new class.

3. If the conflict involves a component explicitly declared in the body of the current class definition and an ancestor of that new class, the new definition is used, which supersedes all ancestor definitions. This is the case even if the type of the new component declaration conflicts with the type declaration of a component of the same name in an ancestor class.

   Note that if the user wants to supersede a function definition from a parent class, that function must be declared in the *body* of the new class as well as being redefined as described below.

2.4 Defining class functions

1) *Declaration*

Different classes may have member functions with the same name. Similarly, a derived class may redefine a function which was defined previously in an ancestor class. It is therefore necessary to have a means to distinguish which of the several possible member functions is being defined. This is done with the class name and the function name joined with the scoping operator "::". So to define the member function "print()" of class "Bishop" which takes a char * argument "str" and returns an integer –

```
int Bishop::print( str )
char * str;
{
        (Body of definition)
}
```

2) *Body*

The body of a member function definition is like the body of an ordinary function body in ClassC, except in the treatment of the components of the object to which the message/function is being sent.

Specifically, the body of a member function definition can refer to other elements of the receiving object without explicitly dereferencing them. The appropriate dereferences are implicit in the function itself. For example:

```
class Piece
        { char * name;
          object position;
          int           checkPosition();
          int           show();
          void          move();
          void          report();
                        /* Send message to coordinator */
        };
```

is a declaration for the class piece. The member function "move()" of class Piece might be defined as follows:

```
void Piece::move( newPosition, board )
object newPosition;
object board;
{   if (checkPosition (newPosition ) )
    {
        board:>atput(newPosition, self);
      position = newPosition;
    }
    else report ("Illegal Move");
}
```

Note the use of the identifier *self* in the definition above. Self is an object variable which is automatically generated by the parser as an argument to the member function. It represents the target object to which the function/message is being sent. The variable name "position" used in the definition of "Piece::move()" above is known to be a component of class Piece, and is therefore assumed to refer to the target object. The statement

"position = newPosition;"

could have been equivalently rewritten

"self :> position = newPosition;"

In general, the "self" variable is used when the member function manipulates the *entire* object rather than the components of that object. In this case, for example, it is used to register the instance of the Piece class to its new position with the object "board".

2.5 Declaring and Initializing Objects

Objects can be declared in one of two ways:

1) By the type name *object,* or

2) By the name of a class defined elsewhere in the file.

When an object variable is declared by a class name, all operations subsequently performed on that variable are checked for conformance at compile time. That is, operations are checked to ensure that they are compatible with the object's declared class or ancestor classes, and the returned result of the message is cast to the appropriate type.

Variables that are just declared to be of type *object* are not subject to any class conformance checking until runtime. Thus, any object operation may be applied to variables declared to be of type *object*. This gives the user more flexibility in the manipulation of the object, but at the risk of runtime errors if the user has used the object incorrectly.

It is suggested that use of the type name *object* be restricted to such applications as collection classes, where it is desirable to allow objects of all types to be stored.

These object type names are syntactically equivalent to standard C scalar type names like "int", "char", "float", etc. This means that they can be used not only to declare variables of type *object*, but arrays of, pointers to, and functions returning *object* as well. So some possible object declarations:

```
object  x;
MyClass       y[4];
object  (*z)[3][7];
Bishop  func();
```

A new object variable is essentially an empty slot into which an object may be placed. Objects themselves are created dynamically from the heap at runtime by a special call or message called "new()". The message "new()" is sent to a class name to create a new instance of that class. This new instance may then be assigned to an object variable (be put in the slot).

Note: Classes in ClassC are not themselves objects, but are syntacticly treated as such with the

"new()" message.

So, for example, to make a new object of class Bishop you write "Bishop:>new()" where the symbol ":>" is the message passing or dereferencing operator of ClassC.

An object variable may be initialized on its declaration line like any other C variable. So, to assign the new instance of Bishop to the object variable *mypiece* we could write either:

```
Bishop mypiece;
mypiece = Bishop:>new();
```

or

```
Bishop mypiece = Bishop :>new();
```

It is also possible to create a special "init()" function for a class which is automatically called when a new object of that class is created. The "init()" function is declared and defined like any other class member function except that it must not be declared to be of any type or return any value. An "init()" function may be defined to take arguments, in which case the call to

"*ClassName*:>new(arg1,arg2,...)"

is made with the appropriate arguments.

New objects created from classes that have no "init()" function associated with them are created with uninitialized memory.

2.6 Deleting Objects

Garbage collection is performed automatically when the ClassC "-G" compiler option is used. The user can also explicitly delete unwanted objects and release their memory space to the heap.

An object is deleted by sending it the message "delete()" –

"oldObj:>delete();"

The function "delete()" does not have to be declared as a component of an object to be used to delete an object. It can be used to delete any object, and just releases the memory specifically allocated to that object by the call to "new()".

It is possible, however, that some objects may have acquired additional memory resources. For instance, an object may contain a pointer to a dynamically allocated string. If the object is deleted without freeing the string, the string will remain, uncollected, floating in memory.

It is thus possible to explicitly define a member function "delete()" as part of a class definition. This member "delete()" function is defined like any class member function, and the user is responsible for cleaning up any additional memory allocated to the object here. When the message "delete()" is sent to an object (possibly with arguments) the user defined member "delete()" function is called to clean up user allocated memory, then the system "_delete()" function is called to delete the object itself.

2.7 Using Objects

There are three different ways in which object variables can be used.

1. They can be used with the dereferencing (component selection) operator ":>" – e.g., "mypiece:>position = newPosition;" or "mypiece:>set(newPosition);"
2. They can be used in assignment statements with other objects or functions returning objects; e.g., "obj1 = obj2;".
3. They can be passed as arguments to functions; e.g. "newobj = transform(oldobj);".

Dereferencing Objects

Member Data:

The operator ":>" is used to dereference or select components of an object. If this component is a data component, the expression

"*object_name :> component_name*"

is an l-value; that is, the expression can be used on either side of the "=" (assignment) operator.

Components of a class can be of any legal C type, or a class C object. If the object variable has been declared with a class name, component type interpretation is done automatically. This means that if object "obj1" has been declared to be of class SampClass and component "mycomp" is an array of floats, the assignment

"obj1:>mycomp [6] = 2.718;"

has the expected result. But as discussed above, object variables declared to be of type *object* are dynamically bound and their type cannot be determined at compile time. Thus, there is no way to determine the type of component "mycomp" at compile time.

Example:
Consider two classes, ClassA and ClassB. Each class has a component named "size". The declaration of "size" for class ClassB "int size;" for ClassA it is "float size;". If the object variable "objVar" is declared to be of type *object,* it is not bound to any class type at compile time and the compiler has no way to know the type of the expression "objVar:>size". This is resolved in the following way:

1. The default return type of the ":>" operator is type "object". Thus, if an object is expected, no special treatment is required.

2. If an abstract declaration appears between the component selector and the component, the type of the expression is cast to the type of the abstract declaration. The abstract declaration is basically a C cast except that the conversion is *exact.* That is, if the component is an array of 6 floats, the abstract declaration for that component MUST be (float [6]). In C, the appropriate cast would be (float *); the cast (float [6]) is illegal in standard C. Similarly, if the desired component is a structure, it must be cast as the structure rather than a pointer to

structure. So if the expected return type of "objVar:>size;" is float, it must be expressed as "objVar:>(float)size".

This interface is awkward, but is provided as an additional feature for the user. When an object variable is declared as an instance of a particular class rather than with the keyword *object,* these conversions are performed automatically and need not be of concern to the user.

2.8 Use of Member Functions

Member functions are accessed through the same ":>" selection operator by which member data are selected. A member function can access other member functions within its body; in particular, member functions of a child can access and call functions inherited by that child from its ancestors. Similarly, an inherited function that references function components that have been redefined in the derived class will call the redefined *derived* class function rather than the function for which it was originally defined. As with data components, function component return values are automatically cast to their declared types for object variables declared by a class name, and can be manually cast as described above for object variables declared by the keyword "object".

2.9 Object Assignment

Objects can be assigned to object variables through the standard C assignment operator "=". This means a reference between the object variable and the object itself is established. No new objects are created by this mechanism. For example, if "a" and "b" are both object variables, then the assignment "a = b;" does NOT duplicate the object pointed to by b. Rather, both "a" and "b" now reference the same object. Thus, any changes made to object "b" will be made to object "a" as well. In order to duplicate an object, the user must allocate new memory and initialize it appropriately, usually through a member function. For example, to create a duplicate of object "objGen", one might write

"AClass dupObj = objGen:>copy();"

where "copy()" has been appropriately defined by the user to create, initialize, and return a new

object.

2.10 Using the -G Garbage Collection Option

Garbage collection is useful for large programs that use large amounts of memory, or programs with repeating loops that reassign object variables without explicitly freeing old values. Garbage collection is a feature which allows the programmer to be less rigorous when writing his/her program, but is costly in terms of computer resources and should be used judiciously.

Note that when the garbage collection option is invoked, objects are statically scoped, and hence are not stacked in recursive function calls. Note also that modules compiled with the garbage collection option must not be combined with other modules compiled without it.

Garbage collection takes place only after a certain number of new objects have been allocated, and thereafter only at specified intervals. Both the initial threshold and the frequency can be modified from their default values by the user.

The external integer "CEGthreshold" is the number of new objects that will be allocated before the garbage collection mechanism is called for the first time. On systems with large memory, it often makes sense to allow this number to be quite large before trying any collection. The value can be changed in any function by declaring the variable

"extern int CEGthreshold;"

and setting it to the desired value.

The frequency of the collection is modifiable as well. *Frequency* of collection, *CEGfrequency,* is defined as how many calls to new() are made between collection attempts. It is wasteful to collect the entire object table every time the user wants to allocate a new object. After the threshold size *CEGthreshold* has been reached, collection takes place after each "CEGfrequency" calls to new(), where CEGfrequency is declared as

"extern int CEGfrequency;"

That is, the integer *CEGfrequency* represents how many calls to *new()* are made between each attempt at collection of the object table.

Again, it is only necessary to declare the variable CEGfrequency if it is desired to modify its default value. The value may be changed by assigning a new integer to it.

The default value of CEGthreshold is 1000; default for CEGfrequency is 100.

2.11 Using ClassC/Elaine

The class definitions should in be put at the beginning of the source file, and should occur before any function definitions.

If a large multi-file project is being developed, all class definitions should be made in header files which are *#included* in each file that manipulates objects of those classes. The member functions of each class should be defined only once, and can be defined in any file and linked in at load time.

The *ce* command invokes the ClassC compiler and processes command line options. The only option special to *ce* is the garbage collection option "-G". Other command line options implement standard cc options as described below:

ce command line options

-O

Invoke the C optimizer on the output.

-o *name*

Call the executable file "name".

-g

Compile for debugging.

-p

Compile for profiling.

-G

Create with garbage collection.

-c

Create object file; do not link into executable.

-l*libname*

Search the library "lib*libname*.a" for unresolved references when linking.

Use the "-c" option and the Unix "ar" library archive command to build ClassC libraries.

2.12 Keywords and special symbols

:>	–	Class dereference operator
::	–	Class scoping operator
:	–	Inheritance operator
//	–	ClassC inline comment operator
class	–	Class definition keyword
object	–	Object type name
self	–	Target object in member function definition
_deref	–	ClassC library dereference function
_new	–	ClassC library object creation function
__new	–	ClassC library object creation function
_decRef	–	ClassC library reference count decrement
_incRef	–	ClassC library reference count increment
_delete	–	ClassC library delete function

2.13 Examples of ClassC programs

See appendices.

3. Implementation

The ClassC/Elaine compiler was implemented in a Unix System V environment with the Yacc and Lex Unix language development tools. The input files are first processed by the C preprocessor *cpp*. This combines all the #include .h files, substitutes all the #define's, strips the comments, and inserts appropriate source file/line number information for error reporting. The output is directed to a /tmp file which is then parsed by the ClassC compiler, which creates a converted .c output file. Finally, the standard C compiler is invoked on the generated .c file to produce the executable code or intermediate .o files for separate compilation. The first task was to build a C language parser to make sure that the language would be a proper superset of C (the new C type *enum* was not included). The the parser was then modified incrementally to add the features of ClassC. The ClassC/Elaine system was implemented as follows:

3.1 Statement/Declaration Mixing

In ClassC, statements and declarations can be mixed together within a single block, as long as variables are declared before being used. In standard C all declarations are required at the start of a block before any executable statements. In order to convert the ClassC blocks into C format, the code within each block is separated into two collections – a statement list and a declaration list. This is a recursive procedure, because blocks can be nested and blocks within blocks are themselves statements. So a global variable "block_number" is maintained and the block code is partitioned into the appropriately nested collection of block statements and declarations. At the end of a block, the statements and declarations are re-merged, with all the declarations floated to the top of the block. The order of the declarations is maintained so that references to previous declarations within the same block are valid.

3.2 Inline Comments

All text between "//" and the newline character is an inline comment and is removed from the input file by the lexical analyzer generated by lex.

3.3 Class Definition

A class definition is written in the form:

```
class Class_name : SuperClass_1 : SuperClass_2 : ...
        {       type declarator;
                type declarator;
                type declarator;
                type declarator;
                        .
                        .
                        .
        };
```

where SuperClasses are optional ancestor classes and declarators may be any legal ClassC declaration; particularly, they may be function declarations or object declarations (not definitions). The parser is a two pass parser. The first pass just builds a list of class names from the class definitions in the file. No kind of error checking is done at this stage. The second pass is the true parse, but all the classes defined in the file are now known to the compiler. Thus, a class which is defined anywhere in the file can be used anywhere in the file. This allows forward referencing inside class definitions, so that class A can be defined to contain objects of class B while class B can likewise be defined to contain objects of class A.

When the second pass of the parser encounters a class definition, it first processes all the declarations within the body of the definition. The variable declarations are treated just like structure component declarations and are entered into a symbol table which is maintained for each class definition. The function declarations are put into a separate function symbol table for the class entry. So each class entry has two symbol tables in it – one for the member functions, and one for member variables.

After building the two symbol tables from the class definition section, the parser examines the parent classes of the class being defined. If the parent classes contain new symbols, these symbols are added to the appropriate symbol tables. If a parent class contains a symbol that is already in

one of the symbol tables and they are declared to be the same type, the duplicate declaration is skipped and the next symbol is processed. If a parent class contains a symbol of the same name as one already in a symbol table but of a *different* declared type, an error has occurred and is reported by the parser.

After all the parent classes have been examined and all the symbol tables built, the ClassC compiler creates two pieces of C code. The first piece of code is a structure definition with the same name as the class, except with a prepended underscore. That is, a class called "Garuda" would have an associated structure definition "_Garuda". This structure definition contains all the variable symbol components of the class, both those explicitly declared in the definition section of that class as well as those symbol names inherited from super classes.

The second section of C code produced by the compiler for every class definition is a set of function declarations corresponding to the function components of the class. The declared function names in this section consist of an initial underscore followed by the class name, underscore, and function name. Only those functions explicitly declared in the class definition section are represented here -- the functions which are inherited from ancestor classes are declared with the ancestor classes. For example, the declared Garuda component "double burn();" would be translated into the C declaration "double _Garuda_burn();".

It would be possible to put pointers to member functions directly in the objects structures. These pointers would be difficult to initialize, however, and would add an extra level of indirection to a member function call. There was no obvious compensating benefit for implementing member functions in this way, and so it was not done.

Tables --

Several tables are maintained by the ClassC compiler while parsing a source file. The class table is a list of all the declared classes with associated information about their ancestors and components. A similar table is maintained for structures, and symbol tables contain information about variables and

functions. The class table is described in detail below.

Each class table entry contains two symbol tables (variable and function) containing information about the data members and function members of that class. Each of these component symbol tables is in turn composed of a linked list of structures, where each structure is a single symbol table entry.

Both function and variable symbol tables use the same structure format for each entry. The symbol table entry structure has the following form:

```
struct sym_entry {
                    struct sym_entry *next;
                    char * name;
                    char * spare;
                    char * typestr;
                    int      blk_no;
                    char * declaration;
                    char * abstract_dec;
                   };
```

1. struct sym_entry * next; –

   A pointer to the next entry in the symbol table.

2. char * name; –

   The declared name of the symbol.

3. char * spare; –

   This is used in the function symbol table of class entries. It contains the real name of the function. The real name of a class function is constructed by concatenating the class name in which the function was defined with the declared name of the function itself. Thus for class Student, the member function "printName();" would appear as "printName" in the "name" field above, and as "_Student_printName" in the "spare" field here if that function is declared in the Student class declaration. If, however, the function "printName" is not declared in the "Student" class definition but is rather inherited from a super class such as "Citizen", the

string appearing in the name field would still be "printName", but the string in the spare field would be the name under which the function is defined; that is, "_Citizen_printName".

4. char * typestr; –

The component "typestr" represents the type of the declared variable. This includes not only the basic type of the variable, but also its indirection. The type is represented by a single letter code for the basic type, preceded by character codes indicating the dereferencing necessary to produce the basic type. That is, if the variable VAR is declared as

"int (*VAR[5])();"

the type string for VAR would be

"ppfi"

which is interpreted as array (same as pointer) (p) of pointers (p) to functions (f) returning integer (i).

The basic types are (i) for *long'* s *short'* s *int'* s *unsigned'* s and *char'* s. The character (d) is used for *double'* s and *float'* s; and so on. The only types that require more than a single character code are aggregate types – *struct'* s, *union'* s, and *class'* s. These are represented by the character (s) or (u) or (k) followed by the structure, union or class name. Thus, for the declaration

"struct Sym_entry * table;"

the variable "table" has associated with it the type string

"psSym_entry",

which is interpreted as pointer (p) to structure (s) Sym_entry.

5. int blk_no; – .

Block number – this is not used in the class entry tables, but the same structure format is

used for all symbol tables used by the parser. The block number is used by the parser with the variable declarations in the file itself to maintain scoping information.

6. char * declaration; –

   The character string *declaration*

   keeps an copy of the declaration string used to declare the component variable. This is used later if any additional classes are derived from the class currently defined. If so, this declaration string is reproduced in the derived class.

7. char * abstract_dec; –

   The abstract declaration equivalent to the variable declaration with the variable name removed. It is used for casting values returned by the class dereferencing function "_deref()" to the appropriate type.

The class table itself is made up of a linked list of class entries, where each entry for a class has the form:

```
struct class_entry
        {
            struct class_entry *next;
            char * name;
            struct sym_entry * sym_table;
            struct sym_entry * fnc_tbl;
            char   * ancList; /* Ancestor list */
            char * initQ;     /* Is there a user defined
                               * init() function for this class?
                               * If so, what is it? */
        } ;
```

1. struct class_entry * next; –

   Pointer to the next entry in the class table.

2. char * name; –

The class name.

3. struct sym_entry * sym_table; --

   The component variable symbol table for the class.

4. struct sym_entry * fnc_tbl; --

   The component function symbol table for the class.

5. char * ancList; --

   The list of superclasses for the class. This list includes not just the super classes declared in the super class declaration line of the class definition, but all the super classes of those super classes as well -- that is, the entire class membership hierarchy for that class.

6. char * initQ;

   Does this class have a declared "init()" function? This is used by the compiler when it encounters a "*ClassName* :>new()" expression for a class. The first thing the "*ClassName* :>new()" expression does is allocate space for the structure defined for *ClassName*. Then, if the user has declared an initialization function for that class, it is called with the appropriate arguments. If no such function has been declared for that class, the structure is allocated with uninitialized memory.

3.4 Inheritance Mechanism

The heart of the ClassC/Elaine system is the inheritance/dereferencing mechanism. This is the most difficult factor in implementing a multiple inheritance object oriented language on top of C.

In C++, ObjectiveC, and ClassC, objects are implemented through C structures and associated functions. This works well for single inheritance systems, because the class components can be laid down in a conventional manner to take advantage of the standard C mechanism for structure dereference.

For example, consider how a linear (single parent line) inheritance system might be implemented. A class declared without any super classes would be represented by a structure containing the member

variables of that class along with a set of functions associated with that structure name.

A class derived from this parent class might have additional variable components. The structure representing the derived class would be identical to that of the parent class except for the additional fields added at the end. This means that all the dereferencing mechanisms that can be applied to the parent class can be applied to the derived class as well because the two classes have the identical structure components in the identical locations for all components found in the parent class. This allows all functions defined for the parent class to be applied to the derived class, which is essential for the concept of class inheritance.

Now consider the case of a multiple inheritance system. The above principle can still apply to the first ancestor of a derived class. What about the second? There are two possibilities.

Orthogonal super classes --

If none of the components of the super classes have any element names in common, there is no real conceptual problem; only tricky pointer manipulation by the compiler. Instead of having identical component layout as its ancestors, the components of the derived class differ from its ancestor classes by a constant offset (which is zero for the first ancestor class and progressively larger for each successive super class). When a derived class is manipulated by functions derived from ancestor classes, the relevant offset can be added to a pointer to the derived class object, and again the lerived object can be manipulated transparently by the super class functions which have no nowledge of the other components of the object.

his solution is good in that it implements a form of multiple inheritance while still allows use of the ındard C structure dereferencing operations. It is fast and very simple to implement.

is solution is bad in that is suffers from a lack of generality. The super classes must be ıogonal, which is not always a reasonable requirement. Imagine two classes used in a university ıbase system -

"Class Student;"

and

"Class Employee".

Imagine we wish to create a new class derived from both –

"Class GradAsst : Employee : Student".

Well, from Class Employee, Class GradAsst can inherit "double Salary;" and "int EmployeeNumber;". From Class Student, Class GradAsst can inherit "double GPA;" and "int StudentID;".

What about "char * name;"? This field is found in both Class Employee and Class Student. The Employee functions will manipulate the employee name component; the Student functions will manipulate the student name field, and which parent provides the name field for GradAsst functions is undefined.

One possible solution is to go back and modify all the existing source code for Employee and Student to change all references to name to Employee_name and Student_name. Then GradAsst could add a new component "char * GradAsst_name;". This solves the ambiguity, but it is now impossible to use functions derived from either Employee or Student to manipulate the new entry GradAsst_name"; and the class GradAsst now has two empty fields it will never use; Employee_name and Student_name. Or GradAsst could just use one of the super class name fields; say Student_name. This reduces some waste but does not give us proper abstraction or symmetry.

Worse, consider the problem of incest. That is, imagine the super class UniversityPersonnel. This class could contain all information common to everyone associated with the university – that is, name, birthdate, address, etc. From class UniversityPersonnel are derived the subclasses "Faculty", "Student", "Staff", "Employee", etc. Then from "Student" and "Employee" we derive "GradAsst" as above. This is an unmanageable situation, which is solved by simply avoiding it. No such

inheritance structure is useful under the multiple inheritance mechanism described above. This is unfortunate, because it is plain that much functionality must now be duplicated in each of the derived classes that could have been handled in the single super class "UniversityPersonnel".

This problem only exists when an attempt is made to achieve multiple inheritance with the standard C structure dereferencing mechanism. This is because at compile time the C compiler converts structure dereference symbols to integer offsets into the structure itself. Therefore the positions of the structure components must be fixed at compile time, which is impossible in a general multiple inheritance scheme. If it were possible to use a different dereferencing mechanism to dereference the structure at *runtime,* the above difficulties in multiple inheritance could be cleanly resolved. The object component would be dereferenced by the symbol itself and not by a precalculated offset. Such a mechanism is used in ClassC. The cost of this mechanism is greatly increased operating complexity and a significant decrease in dereferencing efficiency. This is the mechanism used in ClassC, as described below.

Runtime Symbolic Dereferencing --

After the compiler parses the class definition, it creates a static array of structures "_ref_el". The structure "_ref_el" is a character pointer/integer pair defined as:

```
struct _ref_el    { char * name;
                    int offset;
                  };
```

This data structure is used to match the class component names with their relative position in the class structure. Each class has one such array created for it immediately after the class definition. The array is divided into two sections -- the first section represents the variable components of the class; the second section represents the function components. Each component of the variable section of the array pairs a name of class component with its offset in the structure. The offset is found by casting zero to be a pointer to a structure of the type built for the class, then dereferencing the component for that class, taking its address, and casting the result to an integer.

This is perhaps more clearly illustrated by example:

If the class Munn is declared to have the data component "float eye;" the array of struct _ref_el's for Munn is defined as:

```
struct _ref_el  _Munn_ref [] =
                  { { "eye", (int) (& ( (struct _Munn *) 0)->eye) },
                    { ...
                    { ...
                  };
```

where struct _Munn is the structure created by the parser to represent the variable portion of the class Munn.

This section is terminated by a null character pointer in the name position and the second section depicts the absolute address of the real function corresponding to the class member function. This section is also terminated by a null pointer in the name field.

So for each declared class an array of paired names and offsets is created. This array and the component name is then used by the ClassC runtime function "_deref()" to return the component selected.

The runtime operation is described later; the following example may clarify the above discussion --

Example:

Consider as above class GradAsst as a subclass of both classes Employee and Student. The function printName might be defined for both super classes.

```
class GradAsst : Student : Employee
        { int contract_no;
          printSchedule();
        };
```

Assume that regular employees do not work under grant contracts and hence have no contract number, so the field "contract_no" is unique to class GradAsst. Let us also assume that BOTH class Employee and Student have a member function "printSchedule", but that the format we want for GradAsst is a little different from each, so we intend to redefine it for GradAsst.

The first thing the parser does is make a class entry for the new class GradAsst. It also starts to build two symbol tables for class GradAsst -- one for the variables and one for the functions. The declarations in the body of the GradAsst are processed first and entered into the two symbol tables. Then each symbol entry for Student is scanned by the parser, and compared to the entries already in the class entry for GradAsst. If not already present, a new entry is made in the GradAsst symbol table and the Student symbol entry is copied into it. So GradAsst gets the field "int employeeNumber;" from Employee, and "void printName();" from Student. But since the member function "printSchedule" is included in the definition of GradAsst, the parser ignores the declaration of that function in Student and Employee (except to make sure the type is declared consistently).

After processing the declaration section of the GradAsst definition and building the symbol tables from the superclasses, the ClassC compiler outputs a structure definition for struct "_GradAsst" --

```
struct _GradAsst
        { char * name;
          int contract_no;
          int studentID;
          int employeeNumber;
          struct Schedule schedule;
          .
          .
          .
        };
```

After the structure declaration come the member function declarations:

int _GradAsst_printSchedule();

In this case, since only one member function was declared within the body of the class GradAsst definition, only one function is declared here. Recall that the other inherited functions are declared elsewhere and need not be redeclared here. "printName();", for example, is inherited from class Student, where it is declared "int _Student_printName();". In class GradAsst, the function table contains the entry "printName" which is matched with the full name "_Student_printName" in the field labeled "spare".

After the function declaration section comes the array of "struct _ref_el":

```
struct _ref_el _GradAsst_ref [] =
   { { "name"    , (int) (& ( (struct _GradAsst *) 0)->name) },
     { "contract_no", (int) (&((struct _GradAsst *) 0)->contract_no)},
       .
         .
         .
     { 0,0 }, /* This marks the end of the variable component section*/
     { "printName", (int) _Student_printName },
     { "printSchedule", (int) _GradAsst_printSchedule },
         .
         .
         .
     { 0,0 }, /* This marks the end of the array */
   } };
```

These declarations depend on three features of the C compiler.

1) Variables of arbitrary length are legal and significant.

2) Function names not followed by paired parenthesis are equivalent to a function pointer and return the *address* of the function.

3) Pointers can be represented in unsigned integers.

3.5 Object Table

An object variable in ClassC is implemented as an index into a dynamically constructed object table. Each table entry has three components –

1. The reference count:

   To implement garbage collection, each object table entry has a reference count associated with it. When the garbage collection option is selected by the ClassC command line argument *-G*, the ClassC runtime functions _incRef and _decRef are invoked with each object assignment to keep track of the reference counts.

2. The Variable Structure Pointer:

   All objects of the same class share the identical class functions, but each object has its own private data. As indicated above, this private copy of object data is manipulated in the form of a C structure. When an object is created, a space the size of its data structure is allocated by malloc, and a pointer to that structure is entered into the object table for that object. Since each class has a different structure type, the structure pointer is cast and stored in the object table as an integer.

3. The Class Reference Array:

   When an object is created by invoking "*ClassName* :> new()", an entry is made in the object table and a pointer to the reference array created for *ClassName* is inserted. When the dereferencing function is given an object/index and a selector token (string), it uses the dereferencing array in the object table to find the appropriate offset/address for the token.

3.6 Objects

Objects in ClassC are allocated dynamically from the heap at run time by a call to "new()", which in turn calls the unix function "malloc()" to allocate memory. What the user really sees and manipulates is the *object variable*. Object variables are implemented as *int'* s in C. An object variable is actually an index into the object table described above. When a value is assigned to an object variable, it represents an index into the object table which contains a pointer to the real

object data in memory. Thus, when one object variable is assigned to another, the object data is not copied; just a reference to that object data. THe user must define explicit functions to actually copy the data from one object to another.

3.7 ObjectVariables

When the garbage collection option is selected, objects are implemented as *static* or *extern int'* s in C. The reason for this is reference counting.

Whenever two object expressions appear on each side of an assignment statement, the reference count of the right hand object is incremented and the reference count of the left hand object is decremented BEFORE the assignment is made.

If object variables were created automatically on entry to a function and released again on exit, object references would keep disappearing without the appropriate reference count decrements. Consider the following example:

```
Samp( arg ) object arg;
{ object localObj;
  localObj = arg;
}
```

With the garbage collection option, this ClassC code is translated to:

```
Samp (arg) int arg;
{ static int localObj = 0;
  *_decRef( &localObj ) = _incRef( arg );
}
```

If objects were automatic integer variables, the value of "localObj" is undefined on entry and exit from "Samp()". Thus, "_decRef()" would be applied to an undefined integer; the object pointed to by "arg" would get incremented each time the function was called and some indeterminate entry in the object table would get decremented, or a bus error would occur if "localObj" happened to take on a value out of the range of the table. If "Samp(arg)" were called a thousand times, the object "arg" would be incremented a thousand times and never decremented, while unknown and random

object entries would be decremented. The problem of random decrementing of reference counts could easily be handled by initializing the automatic integer localobj to zero, but the object pointed to by "arg" will still get its reference count incremented each time the function is called, never decrementing the count on exit.

One possible solution to the problem of reference counts would be to keep track of all objects in the function definition and automatically dereference all of them when quitting the function, but functions often have multiple exit points, and it becomes complicated to insert all the reference decrementing code at every possible exit point. In addition, this would add a great deal of runtime overhead to already costly function calls.

Therefore, an alternative solution to the problem of keeping track of object reference counts was found. This solution uses static integers to represent object variables. Static variables are not allocated and deallocated from the stack. Instead they have a continued existence throughout the life of the program. This means that the memory occupied by an object variable is not reclaimed at exit from a function, but this is not as costly as it may seem. Recall that the object *variable* is only an integer pointing to the true (possibly huge) object in dynamic memory. It is the object *variable* which is static in this scheme. In the common case of four bytes per object variable, a program might have several hundred thousand to several million static object variables before the cost of the unreclaimed static integers became significant in a typical application.

These static variables retain their value between calls to a function, thus when the above assignment is performed --

"*_decRef(&localObj) = _incRef(arg);"

the object initially pointed to by localObj is the *previous* object index which was used as an argument on a call to Samp() (or zero, the null object pointer, if this is the first call to Samp()). The reference count of this previous object is now decremented, the reference count of the new argument is incremented, and order is maintained.

All object variables are initialized to zero. _decRef and _incRef recognize the zero index as a special case and take no action when given zero (for _incRef) or a pointer to zero (_decRef) as arguments. Thus, initial assignments can be made to an object variable without decrementing any object reference counts.

Again, this automatic allocation of object variables to static storage classes is only implemented when the -G option is selected at compile time.

3.8 Runtime Library Functions

The ClassC compiler generates function calls to create objects, dereference objects, count references, etc.

3.8.1 The creation functions new()

"new()" creates a new object by allocating memory for the object from the heap, making and initializing a new object entry in the object table, and returning the index of that new entry. There are two "new()" functions; "_new()" (one underscore) and "__new()" (two underscores). The first is used in non-garbage collected environments, the other when garbage collection is implemented.

_new() --

The non-garbage collection version is the simpler of the two. The parser invokes "_new()" when it sees an expression of the form:

"obj = ClassN :> new();".

If class "ClassN" has no "init()" function specified, the parser translates the create object expression to:

"obj = _new (sizeof (struct _ClassN), _ClassN_ref));"

"_new()" first gets the next available index for the object table, and grows the table if necessary. It then allocates space for the data portion of the object through a call to "calloc()" and assigns the

address of the newly allocated memory space to the table entry field "self". The pointer to the array of reference elements (_ClassN) is assigned to the table entry field "class". Finally, the new index is returned and assigned to the object variable "obj".

If the class ClassN *does* have a member "init()" function declared, the translation of the expression

obj = ClassN :> new (argList);"

is

"obj = _ClassN_init(_new(sizeof(struct _ClassN),_ClassN_ref),argList);"

The "_new()" function is invoked in the same way as before, but instead of returning directly to the object "obj", it returns as the first argument to the ClassN "init()" function. The init() function itself operates on the new object with the arguments in the (possibly empty) argument list "argList". When the "_ClassN_init()" function is translated from ClassC to a C function, the ce compiler automatically generates a "return" statement, returning the index of the new object to "obj".

This is why it is illegal for the user to include a "return" statement in an "init()" definition.

__new()

Only the action of "new()" is different when garbage collection is invoked. Collection is an expensive procedure, and should be invoked infrequently. To minimize unnecessary calls to collect, two global (external) variables, CEGthreshold and CEGfrequency, are defined in the ClassC library.

Until the threshold table size is reached, calls to "__new()" execute like calls to the non-garbage collection version. A new table entry is allocated and a new object is created.

After the table size threshold is reached, each call to "__new()" increments a counter. When the counter reaches a multiple of the integer CEGfrequency, the collection routine is invoked.

Both the table space used by an object as well as its actual storage memory are released by the

"collect()" routine. The dynamic storage occupied by the object itself is released by the standard Unix "free()" function. The table entries (indices) occupied by the collected objects are pushed onto a free entry stack.

After the collection, __new() looks at the stack of freed object table indices, pops it, and uses the recycled index for the newly created object.

3.8.2 Garbage collection function 'collect();'

"collect()" is implemented very simply. It starts at the beginning of the object table and examines the reference count field of each entry. If the count is zero (0), "collect()" calls the function "_delete()" on the object, frees the memory allocated to the object, and pushes the object table index number for that object onto the free entry stack, which is popped by "__new()" as described above.

3.8.3 The deletion function '_delete()'

"_delete()" is called automatically by "collect()", or may be called explicitly by the user.

The delete function checks to see if there is a user defined cleanup function "delete()" declared for the class/object being deleted. If so, that cleanup function is called before the object itself is deleted.

For example, an object may contain a character pointer. During the course of the object's lifetime, it may have dynamically allocated a large chunk of memory to which the character pointer now points. Freeing the storage allocated to the object will free the pointer, but not the memory pointed to by the pointer. The user must explicitly define a member "delete()" function for that class that frees the memory pointed to. It is this delete() function that is called by _delete() if it exists.

Whenever "collect()" calls "_delete(obj)", the reference count for object "obj" is zero. This is not necessarily the case if the user explicitly deletes an object. If many object variable reference the same object, one delete should only delete one reference to to object, not the object itself. So the

function "_delete(obj)" first checks the reference count for object "obj". If it is non zero, the object is not deleted; but the reference count is decremented. This is much like the behavior of the Unix "rm" file command.

3.8.4 The reference count functions

"_incRef(obj)" and "_decRef(&obj)" increment and decrement the reference count field in the object table for the object argument.

"_incRef(obj)" simply uses "obj" as an index into the object entry table, increments the associated reference count, and returns the object.

"_decRef(& obj)" takes a pointer to an object as an argument, dereferences it, indexes into the object table, and decrements the appropriate reference count. It returns the object pointer (address).

These function calls are included only when using garbage collection. The expression:

"obj1 = obj2;"

would be translated as:

"* _decRef (&obj1) = _incRef (obj2);"

"_decRef()" takes and returns a pointer argument in order to change the value of the variable "obj1". Furthermore, note that the precedence of the parser is set so that cascaded assignments of objects work as expected. That is:

"obj1 = obj2 = obj3;"

is translated as:

"*_decRef(&obj1)=_incRef (*_decRef (&obj2)= _incref (obj3));"

Object variables were used here to illustrate this example, but this procedure is applied in general to *expressions* that evaluate to class types, of which object variables are just an instance.

For instance, an example of a complex expression that evaluates to an object is

"* someFunction() [6];"

where "someFunction()" is declared to be a function that returns a pointer to an array of object pointers.

3.8.5 The dereferencing function _deref()

The dereferencing operations are the heart of the ClassC system. It is this method that allows transparent dynamic binding and the implementation of multiple inheritance.

1) What does _deref do?
The function "_deref()" takes two arguments. The first argument is an object variable. The second argument is a string/character pointer which is the name of the object component we want to access.

ex -

_deref(objVar, "elementName");

"_deref()" returns a pointer which represents either the address of a data component of the object, or the address of a function component of the object. Recall that all objects of the same class share the same function implementation, but have their own copy of private data. Therefore, "_deref()" must distinguish between function names and data names.

It is necessary to call _deref instead of using a direct structure dereference because of the way multiple inheritance has been implemented in ClassC. An object of a derived class can be assigned to an object variable declared to be of a parent class type. The positions of the components of the derived type may be different from the components of the parent type, and so a standard C

structure dereference (which depends on the order of the components being known at compile time) cannot be used.

2) How is _deref used?

The parser recognizes two types of object dereferences –

a data component dereference that has the form:

"objName:> componentName;"

or a function component dereference of the form:

"objName :> funcName (optionalArgs);"

The compiler translates the data reference expression to a call to _deref() –

_deref(objName,"componentName");

The value returned by "_deref()" is an address. If the compiler knows the class of the object, it looks up the class in the class table to determine what return type to expect from the dereference. It then casts the returned value of _deref() to a pointer to an object of that type, then dereferences the pointer with the pointer operator '*' to generate the actual object itself.

So if "componentName" were a reference to an array of seven integers, the full translation of the reference

"objName :> componentName"

would be

" * (int (*) [7]) _deref(objName,"componentName")"

There are two reasons to have _deref() return an address instead of the actual component dereferenced.

1. A function can only return one type. If _deref() were expected to return the actual entity (component) dereferenced, all components would have to be of the same type. A single dereference function could not be used to return both a structure component and a floating point component. By returning an integer that can be cast as a pointer to any type, we can use a single _deref() function to access any component type.

2. Returning a copy of the component itself would be adequate if we just wanted to read it, but it does not help us if we want to set it. With a pointer implementation, both the following ClassC statements work as expected.

   a = obj :> component;

   obj:> component = a;

   If _deref() returned the value of the actual component itself, the second statement would have no effect; and would in fact translate to an illegal C statement, because the return value of a function is not an l-value.

The same class dereference operator ":>" is used to dereference object *functions*. In this case, _deref() returns a pointer to a function; that is, _deref returns an integer which is immediately cast to a function pointer. In fact, it is cast to a pointer to a function returning the type expected from the component function name. For example, if the component function "nameList()" is expected to return a pointer to an array of character pointers, the reference

obj:>nameList (argList);

will be translated to

((char * (*((*)()))[99]) _deref (obj,"nameList")) (obj, argList)

Thus, _deref() returns an integer which is cast immediately to a pointer to a function which returns

a pointer to an array of pointers to char. This function pointer is then dereferenced by the function pointer dereference operator "(obj, argList)". Note the inclusion of the object name itself in the translated call to the member function "nameList()". This is because the function "nameList()" is meant to access only the private data of the particular object dereferencing it (in this case, the object "obj"); not the data of other objects in the same class. This means that when called, the function "nameList()" must be able to tell which object invoked it. Hence the inclusion of the object variable "obj" in the argument list. The interface is neater if this is done automatically so that the user is free of this responsibility.

3) How is _deref implemented?

The dereference function is called with two arguments -- an object variable (integer index into the object table), and a string (character pointer) representing the field (component) to be dereferenced. The "_deref(object,fieldName)" algorithm proceeds as follows:

1. Get the array of _ref_el for the object class --

   Recall that an object table entry has one field that contains a pointer to an array of string / offset pairs. Vector into the object table with the object index and get the pointer for that array.

2. Do a while loop. Step through the array and examine the value of the character pointer. If the character pointer in the "name" field of the array is NULL (0), this is the end of the data section of the array. The second section of the array has function names, which are dealt with differently by "_deref()". Drop out of the while loop and continue to the next section of the "deref" function (Step 3).

   Otherwise, compare the name field in the array with the string "nameField" passed as an argument to the "_deref()" function.

   1. If the strings are the same, get the integer offset that corresponds to this name in the reference array. Vector into the object table with the object again and get the pointer to the beginning of the structure that holds the object's data. Return the sum of the

offset and the structure address.

2. If the strings are different, increment the index and repeat the loop – go to the next element of the array and repeat the string comparison.

3. If we get this far, we know the fieldName was not found in the first section of the array. Therefore, it was not the name of a data element for that object. Now do the same type of loop as above, checking to see if fieldName is found in the function section of the array. If found this time, however, produce the value of the offset only. This is a pointer to a function, and is not to be added to the address of the object structure. Return it.

4. If we come to the end of the array (another null character pointer in the name field), the component has not been found in the object, and a runtime error is reported.

3.9 The Problem of Non-Idempotentcy

The parser keeps track of the declared type of all identifiers. But an identifier is just a specific instance of the more general syntactic structure "expression". This means that not only variables, but also complicated expressions can evaluate to an object type and then be dereferenced. Thus it is possible to dereference arrays of objects and functions returning objects. For example, if myObj is declared as an array of pointers to object and next() as function returning object, both:

```
*myObj[5]:> myComponent;
```

and

```
next():> reset();
```

are legal ClassC expressions.

This first expression is translated to

```
_deref ( *myObj[5], "myComponent");
```

The second expression poses a problem. Recall that class member functions take the receiver object as the first argument to the function. Thus, the second expression above might be translated to:

```
( (int (*) ()) _deref( next(), "reset" )) ( next());
```

Where the object returned by "next()" is dereferenced with the selector "reset", the value is immediately cast to a function pointer (because "reset" is a member function), and the resulting function pointer is called with the target object returned by "next()" as its only argument.

It is tolerable to call "next()" twice if it always returns the same value. If "next()" is not an idempotent function, however, the program will behave badly.

It is necessary to ensure that a function is evaluated only once in such an expression. This can be done by using a temporary variable. Then the expression can be translated as:

```
((int (*)())_deref(tmpObjVar, "reset" )) (tmpObjVar = next());
```

A problem exists to this solution. There is some ambiguity in the order of evaluation in sequential function expressions. It was empirically discovered that most C compilers evaluate the function arguments from right to left. This is not guaranteed by the language, however, and it is possible that some compilers evaluate from left to right. Therefore, if when the parser itself is built, it can be compiled with the macro REVERSE defined as 1. This will build a parser that reverses the order of assignment above. Thus, if the parser is compiled with REVERSE defined,

```
next():> reset();
```

will be translated to

```
((int (*)())_deref(tmpObjVar = next(), "reset" )) (tmpObjVar);
```

This procedure is automated in the source files for ClassC. Typing "build" at the command line prompt will cause a test file to be compiled and run. The returned result of the test file indicates

whether normal or reverse argument evaluation is implemented in the host C compiler. Then, if the host version of "make" accepts the "-e" (environment) option, "make" will be called and the correct version of the parser will be generated. If make does not accept the "-e" option, the user must simply edit the makefile and define the macro "SWITCH" appropriately ("SWITCH= " for normal compilers; "SWITCH=-DREVERSE" for others).

The only problem now is if there is yet another function dereference:

next():>reset():>double();

Now we need two temporary variables. And so on. This is solved by creating an external array "int indArr[99];", and keeping track of the expression dereferencing depth.

The limitation of this solution is that object function dereferences are restricted to a depth of 99.

4. Implementation

The ClassC/Elaine compiler was implemented in a Unix System V environment with the Yacc and Lex Unix language development tools. The input files are first processed by the C preprocessor *cpp*. This combines all the #include .h files, substitutes all the #define's, strips the comments, and inserts appropriate source file/line number information for error reporting. The output is directed to a /tmp file which is then parsed by the ClassC compiler, which creates a converted .c output file. Finally, the standard C compiler is invoked on the generated .c file to produce the executable code or intermediate .o files for separate compilation. The first task was to build a C language parser to make sure that the language would be a proper superset of C (the new C type *enum* was not included). The the parser was then modified incrementally to add the features of ClassC. The ClassC/Elaine system was implemented as follows:

4.1 Statement/Declaration Mixing

In ClassC, statements and declarations can be mixed together within a single block, as long as variables are declared before being used. In standard C all declarations are required at the start of a block before any executable statements. In order to convert the ClassC blocks into C format, the code within each block is separated into two collections – a statement list and a declaration list. This is a recursive procedure, because blocks can be nested and blocks within blocks are themselves statements. So a global variable "block_number" is maintained and the block code is partitioned into the appropriately nested collection of block statements and declarations. At the end of a block, the statements and declarations are re-merged, with all the declarations floated to the top of the block. The order of the declarations is maintained so that references to previous declarations within the same block are valid.

4.2 Inline Comments

All text between "//" and the newline character is an inline comment and is removed from the input file by the lexical analyzer generated by lex.

4.3 Class Definition

A class definition is written in the form:

```
class Class_name : SuperClass_1 : SuperClass_2 : ...
        {       type declarator;
                type declarator;
                type declarator;
                type declarator;
                        .
                        .
                        .
        };
```

where SuperClasses are optional ancestor classes and declarators may be any legal ClassC declaration; particularly, they may be function declarations or object declarations (not definitions). The parser is a two pass parser. The first pass just builds a list of class names from the class definitions in the file. No kind of error checking is done at this stage. The second pass is the true parse, but all the classes defined in the file are now known to the compiler. Thus, a class which is defined anywhere in the file can be used anywhere in the file. This allows forward referencing inside class definitions, so that class A can be defined to contain objects of class B while class B can likewise be defined to contain objects of class A.

When the second pass of the parser encounters a class definition, it first processes all the declarations within the body of the definition. The variable declarations are treated just like structure component declarations and are entered into a symbol table which is maintained for each class definition. The function declarations are put into a separate function symbol table for the class entry. So each class entry has two symbol tables in it – one for the member functions, and one for member variables.

After building the two symbol tables from the class definition section, the parser examines the parent classes of the class being defined. If the parent classes contain new symbols, these symbols are added to the appropriate symbol tables. If a parent class contains a symbol that is already in

one of the symbol tables and they are declared to be the same type, the duplicate declaration is skipped and the next symbol is processed. If a parent class contains a symbol of the same name as one already in a symbol table but of a *different* declared type, an error has occurred and is reported by the parser.

After all the parent classes have been examined and all the symbol tables built, the ClassC compiler creates two pieces of C code. The first piece of code is a structure definition with the same name as the class, except with a prepended underscore. That is, a class called "Garuda" would have an associated structure definition "_Garuda". This structure definition contains all the variable symbol components of the class, both those explicitly declared in the definition section of that class as well as those symbol names inherited from super classes.

The second section of C code produced by the compiler for every class definition is a set of function declarations corresponding to the function components of the class. The declared function names in this section consist of an initial underscore followed by the class name, underscore, and function name. Only those functions explicitly declared in the class definition section are represented here -- the functions which are inherited from ancestor classes are declared with the ancestor classes. For example, the declared Garuda component "double burn();" would be translated into the C declaration "double _Garuda_burn();".

It would be possible to put pointers to member functions directly in the objects structures. These pointers would be difficult to initialize, however, and would add an extra level of indirection to a member function call. There was no obvious compensating benefit for implementing member functions in this way, and so it was not done.

Tables --

Several tables are maintained by the ClassC compiler while parsing a source file. The class table is a list of all the declared classes with associated information about their ancestors and components. A similar table is maintained for structures, and symbol tables contain information about variables and

functions. The class table is described in detail below.

Each class table entry contains two symbol tables (variable and function) containing information about the data members and function members of that class. Each of these component symbol tables is in turn composed of a linked list of structures, where each structure is a single symbol table entry.

Both function and variable symbol tables use the same structure format for each entry. The symbol table entry structure has the following form:

```
struct sym_entry {
                    struct sym_entry *next;
                    char * name;
                    char * spare;
                    char * typestr;
                    int      blk_no;
                    char * declaration;
                    char * abstract_dec;
                  };
```

1. struct sym_entry * next; –

   A pointer to the next entry in the symbol table.

2. char * name; –

   The declared name of the symbol.

3. char * spare; –

   This is used in the function symbol table of class entries. It contains the real name of the function. The real name of a class function is constructed by concatenating the class name in which the function was defined with the declared name of the function itself. Thus for class Student, the member function "printName();" would appear as "printName" in the "name" field above, and as "_Student_printName" in the "spare" field here if that function is declared in the Student class declaration. If, however, the function "printName" is not declared in the "Student" class definition but is rather inherited from a super class such as "Citizen", the

string appearing in the name field would still be "printName", but the string in the spare field would be the name under which the function is defined; that is, "_Citizen_printName".

4. char * typestr; –

The component "typestr" represents the type of the declared variable. This includes not only the basic type of the variable, but also its indirection. The type is represented by a single letter code for the basic type, preceded by character codes indicating the dereferencing necessary to produce the basic type. That is, if the variable VAR is declared as

"int (*VAR[5])();"

the type string for VAR would be

"ppfi"

which is interpreted as array (same as pointer) (p) of pointers (p) to functions (f) returning integer (i).

The basic types are (i) for *long'* s *short'* s *int'* s *unsigned'* s and *char'* s. The character (d) is used for *double'* s and *float'* s; and so on. The only types that require more than a single character code are aggregate types – *struct'* s, *union'* s, and *class'* s. These are represented by the character (s) or (u) or (k) followed by the structure, union or class name. Thus, for the declaration

"struct Sym_entry * table;"

the variable "table" has associated with it the type string

"psSym_entry",

which is interpreted as pointer (p) to structure (s) Sym_entry.

5. int blk_no; –

Block number – this is not used in the class entry tables, but the same structure format is

used for all symbol tables used by the parser. The block number is used by the parser with the variable declarations in the file itself to maintain scoping information.

6. char * declaration; –

   The character string *declaration*

   keeps an copy of the declaration string used to declare the component variable. This is used later if any additional classes are derived from the class currently defined. If so, this declaration string is reproduced in the derived class.

7. char * abstract_dec; –

   The abstract declaration equivalent to the variable declaration with the variable name removed. It is used for casting values returned by the class dereferencing function "_deref()" to the appropriate type.

The class table itself is made up of a linked list of class entries, where each entry for a class has the form:

```
struct class_entry
        {
            struct class_entry *next;
            char * name;
            struct sym_entry * sym_table;
            struct sym_entry * fnc_tbl;
            char   * ancList; /* Ancestor list */
            char * initQ;     /* Is there a user defined
                               * init() function for this class?
                               * If so, what is it? */
        } ;
```

1. struct class_entry * next; –

   Pointer to the next entry in the class table.

2. char * name; –

The class name.

3. struct sym_entry * sym_table; –

   The component variable symbol table for the class.

4. struct sym_entry * fnc_tbl; –

   The component function symbol table for the class.

5. char * ancList; –

   The list of superclasses for the class. This list includes not just the super classes declared in the super class declaration line of the class definition, but all the super classes of those super classes as well – that is, the entire class membership hierarchy for that class.

6. char * initQ;

   Does this class have a declared "init()" function? This is used by the compiler when it encounters a "*ClassName* :>new()" expression for a class. The first thing the "*ClassName* :>new()" expression does is allocate space for the structure defined for *ClassName*. Then, if the user has declared an initialization function for that class, it is called with the appropriate arguments. If no such function has been declared for that class, the structure is allocated with uninitialized memory.

4.4 Inheritance Mechanism

The heart of the ClassC/Elaine system is the inheritance/dereferencing mechanism. This is the most difficult factor in implementing a multiple inheritance object oriented language on top of C.

In C++, ObjectiveC, and ClassC, objects are implemented through C structures and associated functions. This works well for single inheritance systems, because the class components can be laid down in a conventional manner to take advantage of the standard C mechanism for structure dereference.

For example, consider how a linear (single parent line) inheritance system might be implemented. A class declared without any super classes would be represented by a structure containing the member

variables of that class along with a set of functions associated with that structure name.

A class derived from this parent class might have additional variable components. The structure representing the derived class would be identical to that of the parent class except for the additional fields added at the end. This means that all the dereferencing mechanisms that can be applied to the parent class can be applied to the derived class as well because the two classes have the identical structure components in the identical locations for all components found in the parent class. This allows all functions defined for the parent class to be applied to the derived class, which is essential for the concept of class inheritance.

Now consider the case of a multiple inheritance system. The above principle can still apply to the first ancestor of a derived class. What about the second? There are two possibilities.

Orthogonal super classes --

If none of the components of the super classes have any element names in common, there is no real conceptual problem; only tricky pointer manipulation by the compiler. Instead of having identical component layout as its ancestors, the components of the derived class differ from its ancestor classes by a constant offset (which is zero for the first ancestor class and progressively larger for each successive super class). When a derived class is manipulated by functions derived from ancestor classes, the relevant offset can be added to a pointer to the derived class object, and again the derived object can be manipulated transparently by the super class functions which have no knowledge of the other components of the object.

This solution is good in that it implements a form of multiple inheritance while still allows use of the standard C structure dereferencing operations. It is fast and very simple to implement.

This solution is bad in that is suffers from a lack of generality. The super classes must be orthogonal, which is not always a reasonable requirement. Imagine two classes used in a university database system -

"Class Student;"

and

"Class Employee".

Imagine we wish to create a new class derived from both --

"Class GradAsst : Employee : Student".

Well, from Class Employee, Class GradAsst can inherit "double Salary;" and "int EmployeeNumber;". From Class Student, Class GradAsst can inherit "double GPA;" and "int StudentID;".

What about "char * name;"? This field is found in both Class Employee and Class Student. The Employee functions will manipulate the employee name component; the Student functions will manipulate the student name field, and which parent provides the name field for GradAsst functions is undefined.

One possible solution is to go back and modify all the existing source code for Employee and Student to change all references to name to Employee_name and Student_name. Then GradAsst could add a new component "char * GradAsst_name;". This solves the ambiguity, but it is now impossible to use functions derived from either Employee or Student to manipulate the new entry GradAsst_name"; and the class GradAsst now has two empty fields it will never use; Employee_name and Student_name. Or GradAsst could just use one of the super class name fields; say Student_name. This reduces some waste but does not give us proper abstraction or symmetry.

Worse, consider the problem of incest. That is, imagine the super class UniversityPersonnel. This class could contain all information common to everyone associated with the university -- that is, name, birthdate, address, etc. From class UniversityPersonnel are derived the subclasses "Faculty", "Student", "Staff", "Employee", etc. Then from "Student" and "Employee" we derive "GradAsst" as above. This is an unmanageable situation, which is solved by simply avoiding it. No such

inheritance structure is useful under the multiple inheritance mechanism described above. This is unfortunate, because it is plain that much functionality must now be duplicated in each of the derived classes that could have been handled in the single super class "UniversityPersonnel".

This problem only exists when an attempt is made to achieve multiple inheritance with the standard C structure dereferencing mechanism. This is because at compile time the C compiler converts structure dereference symbols to integer offsets into the structure itself. Therefore the positions of the structure components must be fixed at compile time, which is impossible in a general multiple inheritance scheme. If it were possible to use a different dereferencing mechanism to dereference the structure at *runtime,* the above difficulties in multiple inheritance could be cleanly resolved. The object component would be dereferenced by the symbol itself and not by a precalculated offset. Such a mechanism is used in ClassC. The cost of this mechanism is greatly increased operating complexity and a significant decrease in dereferencing efficiency. This is the mechanism used in ClassC, as described below.

Runtime Symbolic Dereferencing –

After the compiler parses the class definition, it creates a static array of structures "_ref_el". The structure "_ref_el" is a character pointer/integer pair defined as:

```
struct _ref_el    { char * name;
                    int offset;
                  };
```

This data structure is used to match the class component names with their relative position in the class structure. Each class has one such array created for it immediately after the class definition. The array is divided into two sections – the first section represents the variable components of the class; the second section represents the function components. Each component of the variable section of the array pairs a name of class component with its offset in the structure. The offset is found by casting zero to be a pointer to a structure of the type built for the class, then dereferencing the component for that class, taking its address, and casting the result to an integer.

This is perhaps more clearly illustrated by example:

If the class Munn is declared to have the data component "float eye;" the array of struct _ref_el's for Munn is defined as:

```
struct _ref_el  _Munn_ref [] =
                  { { "eye", (int) (& ( (struct _Munn *) 0)->eye) },
                    { ...
                    { ...
                  };
```

where struct _Munn is the structure created by the parser to represent the variable portion of the class Munn.

This section is terminated by a null character pointer in the name position and the second section depicts the absolute address of the real function corresponding to the class member function. This section is also terminated by a null pointer in the name field.

So for each declared class an array of paired names and offsets is created. This array and the component name is then used by the ClassC runtime function "_deref()" to return the component selected.

The runtime operation is described later; the following example may clarify the above discussion --

Example:

Consider as above class GradAsst as a subclass of both classes Employee and Student. The function printName might be defined for both super classes.

```
class GradAsst : Student : Employee
        { int contract_no;
          printSchedule();
        };
```

Assume that regular employees do not work under grant contracts and hence have no contract number, so the field "contract_no" is unique to class GradAsst. Let us also assume that BOTH class Employee and Student have a member function "printSchedule", but that the format we want for GradAsst is a little different from each, so we intend to redefine it for GradAsst.

The first thing the parser does is make a class entry for the new class GradAsst. It also starts to build two symbol tables for class GradAsst -- one for the variables and one for the functions. The declarations in the body of the GradAsst are processed first and entered into the two symbol tables. Then each symbol entry for Student is scanned by the parser, and compared to the entries already in the class entry for GradAsst. If not already present, a new entry is made in the GradAsst symbol table and the Student symbol entry is copied into it. So GradAsst gets the field "int employeeNumber;" from Employee, and "void printName();" from Student. But since the member function "printSchedule" is included in the definition of GradAsst, the parser ignores the declaration of that function in Student and Employee (except to make sure the type is declared consistently).

After processing the declaration section of the GradAsst definition and building the symbol tables from the superclasses, the ClassC compiler outputs a structure definition for struct "_GradAsst" --

```
struct _GradAsst
        { char * name;
          int contract_no;
          int studentID;
          int employeeNumber;
          struct Schedule schedule;
          .
          .
          .
        };
```

After the structure declaration come the member function declarations:

int _GradAsst_printSchedule();

In this case, since only one member function was declared within the body of the class GradAsst definition, only one function is declared here. Recall that the other inherited functions are declared elsewhere and need not be redeclared here. "printName();", for example, is inherited from class Student, where it is declared "int _Student_printName();". In class GradAsst, the function table contains the entry "printName" which is matched with the full name "_Student_printName" in the field labeled "spare".

After the function declaration section comes the array of "struct _ref_el":

```
struct _ref_el _GradAsst_ref [] =
   { { "name"    , (int) (& ( (struct _GradAsst *) 0)->name) },
     { "contract_no", (int) (&((struct _GradAsst *) 0)->contract_no)},
       .
          .
          .
     { 0,0 }, /* This marks the end of the variable component section*/
     { "printName", (int) _Student_printName },
     { "printSchedule", (int) _GradAsst_printSchedule },
          .
          .
          .
     { 0,0 }, /* This marks the end of the array */
   } };
```

These declarations depend on three features of the C compiler.

1) Variables of arbitrary length are legal and significant.

2) Function names not followed by paired parenthesis are equivalent to a function pointer and return the *address* of the function.

3) Pointers can be represented in unsigned integers.

4.5 Object Table

An object variable in ClassC is implemented as an index into a dynamically constructed object table. Each table entry has three components –

1. The reference count:

   To implement garbage collection, each object table entry has a reference count associated with it. When the garbage collection option is selected by the ClassC command line argument *-G*, the ClassC runtime functions _incRef and _decRef are invoked with each object assignment to keep track of the reference counts.

2. The Variable Structure Pointer:

   All objects of the same class share the identical class functions, but each object has its own private data. As indicated above, this private copy of object data is manipulated in the form of a C structure. When an object is created, a space the size of its data structure is allocated by malloc, and a pointer to that structure is entered into the object table for that object. Since each class has a different structure type, the structure pointer is cast and stored in the object table as an integer.

3. The Class Reference Array:

   When an object is created by invoking "*ClassName* :> new()", an entry is made in the object table and a pointer to the reference array created for *ClassName* is inserted. When the dereferencing function is given an object/index and a selector token (string), it uses the dereferencing array in the object table to find the appropriate offset/address for the token.

4.6 Objects

Objects in ClassC are allocated dynamically from the heap at run time by a call to "new()", which in turn calls the unix function "malloc()" to allocate memory. What the user really sees and manipulates is the *object variable*. Object variables are implemented as *int'* s in C. An object variable is actually an index into the object table described above. When a value is assigned to an object variable, it represents an index into the object table which contains a pointer to the real

object data in memory. Thus, when one object variable is assigned to another, the object data is not copied; just a reference to that object data. THe user must define explicit functions to actually copy the data from one object to another.

4.7 ObjectVariables

When the garbage collection option is selected, objects are implemented as *static* or *extern int'* s in C. The reason for this is reference counting.

Whenever two object expressions appear on each side of an assignment statement, the reference count of the right hand object is incremented and the reference count of the left hand object is decremented BEFORE the assignment is made.

If object variables were created automatically on entry to a function and released again on exit, object references would keep disappearing without the appropriate reference count decrements. Consider the following example:

```
Samp( arg ) object arg;
{ object localObj;
  localObj = arg;
}
```

With the garbage collection option, this ClassC code is translated to:

```
Samp (arg) int arg;
{ static int localObj = 0;
  *_decRef( &localObj ) = _incRef( arg );
}
```

If objects were automatic integer variables, the value of "localObj" is undefined on entry and exit from "Samp()". Thus, "_decRef()" would be applied to an undefined integer; the object pointed to by "arg" would get incremented each time the function was called and some indeterminate entry in the object table would get decremented, or a bus error would occur if "localObj" happened to take on a value out of the range of the table. If "Samp(arg)" were called a thousand times, the object "arg" would be incremented a thousand times and never decremented, while unknown and random

object entries would be decremented. The problem of random decrementing of reference counts could easily be handled by initializing the automatic integer localobj to zero, but the object pointed to by "arg" will still get its reference count incremented each time the function is called, never decrementing the count on exit.

One possible solution to the problem of reference counts would be to keep track of all objects in the function definition and automatically dereference all of them when quitting the function, but functions often have multiple exit points, and it becomes complicated to insert all the reference decrementing code at every possible exit point. In addition, this would add a great deal of runtime overhead to already costly function calls.

Therefore, an alternative solution to the problem of keeping track of object reference counts was found. This solution uses static integers to represent object variables. Static variables are not allocated and deallocated from the stack. Instead they have a continued existence throughout the life of the program. This means that the memory occupied by an object variable is not reclaimed at exit from a function, but this is not as costly as it may seem. Recall that the object *variable* is only an integer pointing to the true (possibly huge) object in dynamic memory. It is the object *variable* which is static in this scheme. In the common case of four bytes per object variable, a program might have several hundred thousand to several million static object variables before the cost of the unreclaimed static integers became significant in a typical application.

These static variables retain their value between calls to a function, thus when the above assignment is performed –

"*_decRef(&localObj) = _incRef(arg);"

the object initially pointed to by localObj is the *previous* object index which was used as an argument on a call to Samp() (or zero, the null object pointer, if this is the first call to Samp()). The reference count of this previous object is now decremented, the reference count of the new argument is incremented, and order is maintained.

All object variables are initialized to zero. _decRef and _incRef recognize the zero index as a special case and take no action when given zero (for _incRef) or a pointer to zero (_decRef) as arguments. Thus, initial assignments can be made to an object variable without decrementing any object reference counts.

Again, this automatic allocation of object variables to static storage classes is only implemented when the -G option is selected at compile time.

4.8 Runtime Library Functions

The ClassC compiler generates function calls to create objects, dereference objects, count references, etc.

4.8.1 The creation functions new()

"new()" creates a new object by allocating memory for the object from the heap, making and initializing a new object entry in the object table, and returning the index of that new entry. There are two "new()" functions; "_new()" (one underscore) and "__new()" (two underscores). The first is used in non-garbage collected environments, the other when garbage collection is implemented.

_new() --

The non-garbage collection version is the simpler of the two. The parser invokes "_new()" when it sees an expression of the form:

"obj = ClassN :> new();".

If class "ClassN" has no "init()" function specified, the parser translates the create object expression to:

"obj = _new (sizeof (struct _ClassN), _ClassN_ref));"

"_new()" first gets the next available index for the object table, and grows the table if necessary. It then allocates space for the data portion of the object through a call to "calloc()" and assigns the

address of the newly allocated memory space to the table entry field "self". The pointer to the array of reference elements (_ClassN) is assigned to the table entry field "class". Finally, the new index is returned and assigned to the object variable "obj".

If the class ClassN *does* have a member "init()" function declared, the translation of the expression

obj = ClassN :> new (argList);"

is

"obj = _ClassN_init(_new(sizeof(struct _ClassN),_ClassN_ref),argList);"

The "_new()" function is invoked in the same way as before, but instead of returning directly to the object "obj", it returns as the first argument to the ClassN "init()" function. The init() function itself operates on the new object with the arguments in the (possibly empty) argument list "argList". When the "_ClassN_init()" function is translated from ClassC to a C function, the ce compiler automatically generates a "return" statement, returning the index of the new object to "obj".

This is why it is illegal for the user to include a "return" statement in an "init()" definition.

__new()

Only the action of "new()" is different when garbage collection is invoked. Collection is an expensive procedure, and should be invoked infrequently. To minimize unnecessary calls to collect, two global (external) variables, CEGthreshold and CEGfrequency, are defined in the ClassC library.

Until the threshold table size is reached, calls to "__new()" execute like calls to the non-garbage collection version. A new table entry is allocated and a new object is created.

After the table size threshold is reached, each call to "__new()" increments a counter. When the counter reaches a multiple of the integer CEGfrequency, the collection routine is invoked.

Both the table space used by an object as well as its actual storage memory are released by the

"collect()" routine. The dynamic storage occupied by the object itself is released by the standard Unix "free()" function. The table entries (indices) occupied by the collected objects are pushed onto a free entry stack.

After the collection, __new() looks at the stack of freed object table indices, pops it, and uses the recycled index for the newly created object.

4.8.2 Garbage collection function 'collect();'

"collect()" is implemented very simply. It starts at the beginning of the object table and examines the reference count field of each entry. If the count is zero (0), "collect()" calls the function "_delete()" on the object, frees the memory allocated to the object, and pushes the object table index number for that object onto the free entry stack, which is popped by "__new()" as described above.

4.8.3 The deletion function '_delete()'

"_delete()" is called automatically by "collect()", or may be called explicitly by the user.

The delete function checks to see if there is a user defined cleanup function "delete()" declared for the class/object being deleted. If so, that cleanup function is called before the object itself is deleted.

For example, an object may contain a character pointer. During the course of the object's lifetime, it may have dynamically allocated a large chunk of memory to which the character pointer now points. Freeing the storage allocated to the object will free the pointer, but not the memory pointed to by the pointer. The user must explicitly define a member "delete()" function for that class that frees the memory pointed to. It is this delete() function that is called by _delete() if it exists.

Whenever "collect()" calls "_delete(obj)", the reference count for object "obj" is zero. This is not necessarily the case if the user explicitly deletes an object. If many object variable reference the same object, one delete should only delete one reference to to object, not the object itself. So the

function "_delete(obj)" first checks the reference count for object "obj". If it is non - zero, the object is not deleted; but the reference count is decremented. This is much like the behavior of the Unix "rm" file command.

4.8.4 The reference count functions

"_incRef(obj)" and "_decRef(&obj)" increment and decrement the reference count field in the object table for the object argument.

"_incRef(obj)" simply uses "obj" as an index into the object entry table, increments the associated reference count, and returns the object.

"_decRef(& obj)" takes a pointer to an object as an argument, dereferences it, indexes into the object table, and decrements the appropriate reference count. It returns the object pointer (address).

These function calls are included only when using garbage collection. The expression:

"obj1 = obj2;"

would be translated as:

"* _decRef (&obj1) = _incRef (obj2);"

"_decRef()" takes and returns a pointer argument in order to change the value of the variable "obj1". Furthermore, note that the precedence of the parser is set so that cascaded assignments of objects work as expected. That is:

"obj1 = obj2 = obj3;"

is translated as:

"*_decRef(&obj1)=_incRef (*_decRef (&obj2)= _incref (obj3));"

Object variables were used here to illustrate this example, but this procedure is applied in general to *expressions* that evaluate to class types, of which object variables are just an instance.

For instance, an example of a complex expression that evaluates to an object is

"* someFunction() [6];"

where "someFunction()" is declared to be a function that returns a pointer to an array of object pointers.

4.8.5 The dereferencing function _deref()

The dereferencing operations are the heart of the ClassC system. It is this method that allows transparent dynamic binding and the implementation of multiple inheritance.

1) What does _deref do?
The function "_deref()" takes two arguments. The first argument is an object variable. The second argument is a string/character pointer which is the name of the object component we want to access.

ex -

_deref(objVar, "elementName");

"_deref()" returns a pointer which represents either the address of a data component of the object, or the address of a function component of the object. Recall that all objects of the same class share the same function implementation, but have their own copy of private data. Therefore, "_deref()" must distinguish between function names and data names.

It is necessary to call _deref instead of using a direct structure dereference because of the way multiple inheritance has been implemented in ClassC. An object of a derived class can be assigned to an object variable declared to be of a parent class type. The positions of the components of the derived type may be different from the components of the parent type, and so a standard C

structure dereference (which depends on the order of the components being known at compile time) cannot be used.

2) How is _deref used?

The parser recognizes two types of object dereferences –

a data component dereference that has the form:

"objName:> componentName;"

or a function component dereference of the form:

"objName :> funcName (optionalArgs);"

The compiler translates the data reference expression to a call to _deref() –

_deref(objName,"componentName");

The value returned by "_deref()" is an address. If the compiler knows the class of the object, it looks up the class in the class table to determine what return type to expect from the dereference. It then casts the returned value of _deref() to a pointer to an object of that type, then dereferences the pointer with the pointer operator '*' to generate the actual object itself.

So if "componentName" were a reference to an array of seven integers, the full translation of the reference

"objName :> componentName"

would be

" * (int (*) [7]) _deref(objName,"componentName")"

There are two reasons to have _deref() return an address instead of the actual component dereferenced.

1. A function can only return one type. If _deref() were expected to return the actual entity (component) dereferenced, all components would have to be of the same type. A single dereference function could not be used to return both a structure component and a floating point component. By returning an integer that can be cast as a pointer to any type, we can use a single _deref() function to access any component type.

2. Returning a copy of the component itself would be adequate if we just wanted to read it, but it does not help us if we want to set it. With a pointer implementation, both the following ClassC statements work as expected.

   a = obj :> component;

   obj:> component = a;

   If _deref() returned the value of the actual component itself, the second statement would have no effect; and would in fact translate to an illegal C statement, because the return value of a function is not an l-value.

The same class dereference operator ":>" is used to dereference object *functions*. In this case, _deref() returns a pointer to a function; that is, _deref returns an integer which is immediately cast to a function pointer. In fact, it is cast to a pointer to a function returning the type expected from the component function name. For example, if the component function "nameList()" is expected to return a pointer to an array of character pointers, the reference

obj:>nameList (argList);

will be translated to

((char * (*((*)()))[99]) _deref (obj,"nameList")) (obj, argList)

Thus, _deref() returns an integer which is cast immediately to a pointer to a function which returns

a pointer to an array of pointers to char. This function pointer is then dereferenced by the function pointer dereference operator "(obj, argList)". Note the inclusion of the object name itself in the translated call to the member function "nameList()". This is because the function "nameList()" is meant to access only the private data of the particular object dereferencing it (in this case, the object "obj"); not the data of other objects in the same class. This means that when called, the function "nameList()" must be able to tell which object invoked it. Hence the inclusion of the object variable "obj" in the argument list. The interface is neater if this is done automatically so that the user is free of this responsibility.

3) How is _deref implemented?

The dereference function is called with two arguments – an object variable (integer index into the object table), and a string (character pointer) representing the field (component) to be dereferenced. The "_deref(object,fieldName)" algorithm proceeds as follows:

1. Get the array of _ref_el for the object class –

   Recall that an object table entry has one field that contains a pointer to an array of string / offset pairs. Vector into the object table with the object index and get the pointer for that array.

2. Do a while loop. Step through the array and examine the value of the character pointer. If the character pointer in the "name" field of the array is NULL (0), this is the end of the data section of the array. The second section of the array has function names, which are dealt with differently by "_deref()". Drop out of the while loop and continue to the next section of the "deref" function (Step 3).

   Otherwise, compare the name field in the array with the string "nameField" passed as an argument to the "_deref()" function.

   1. If the strings are the same, get the integer offset that corresponds to this name in the reference array. Vector into the object table with the object again and get the pointer to the beginning of the structure that holds the object's data. Return the sum of the

offset and the structure address.

2. If the strings are different, increment the index and repeat the loop -- go to the next element of the array and repeat the string comparison.

3. If we get this far, we know the fieldName was not found in the first section of the array. Therefore, it was not the name of a data element for that object. Now do the same type of loop as above, checking to see if fieldName is found in the function section of the array. If found this time, however, produce the value of the offset only. This is a pointer to a function, and is not to be added to the address of the object structure. Return it.

4. If we come to the end of the array (another null character pointer in the name field), the component has not been found in the object, and a runtime error is reported.

4.9 The Problem of Non-Idempotentcy

The parser keeps track of the declared type of all identifiers. But an identifier is just a specific instance of the more general syntactic structure "expression". This means that not only variables, but also complicated expressions can evaluate to an object type and then be dereferenced. Thus it is possible to dereference arrays of objects and functions returning objects. For example, if myObj is declared as an array of pointers to object and next() as function returning object, both:

```
*myObj[5]:> myComponent;
```

and

```
next():> reset();
```

are legal ClassC expressions.

This first expression is translated to

```
_deref ( *myObj[5], "myComponent");
```

The second expression poses a problem. Recall that class member functions take the receiver object as the first argument to the function. Thus, the second expression above might be translated to:

```
( (int (*) ()) _deref( next(), "reset" )) ( next());
```

Where the object returned by "next()" is dereferenced with the selector "reset", the value is immediately cast to a function pointer (because "reset" is a member function), and the resulting function pointer is called with the target object returned by "next()" as its only argument.

It is tolerable to call "next()" twice if it always returns the same value. If "next()" is not an idempotent function, however, the program will behave badly.

It is necessary to ensure that a function is evaluated only once in such an expression. This can be done by using a temporary variable. Then the expression can be translated as:

```
((int (*)())_deref(tmpObjVar, "reset" )) (tmpObjVar = next());
```

A problem exists to this solution. There is some ambiguity in the order of evaluation in sequential function expressions. It was empirically discovered that most C compilers evaluate the function arguments from right to left. This is not guaranteed by the language, however, and it is possible that some compilers evaluate from left to right. Therefore, if when the parser itself is built, it can be compiled with the macro REVERSE defined as 1. This will build a parser that reverses the order of assignment above. Thus, if the parser is compiled with REVERSE defined,

```
next():> reset();
```

will be translated to

```
((int (*)())_deref(tmpObjVar = next(), "reset" )) (tmpObjVar);
```

This procedure is automated in the source files for ClassC. Typing "build" at the command line prompt will cause a test file to be compiled and run. The returned result of the test file indicates

whether normal or reverse argument evaluation is implemented in the host C compiler. Then, if the host version of "make" accepts the "-e" (environment) option, "make" will be called and the correct version of the parser will be generated. If make does not accept the "-e" option, the user must simply edit the makefile and define the macro "SWITCH" appropriately ("SWITCH= " for normal compilers; "SWITCH=-DREVERSE" for others).

The only problem now is if there is yet another function dereference:

next():>reset():>double();

Now we need two temporary variables. And so on. This is solved by creating an external array "int indArr[99];", and keeping track of the expression dereferencing depth.

The limitation of this solution is that object function dereferences are restricted to a depth of 99.

5. *Appendicies -- Sample ClassC Programs*

5.1 ChessMovs.e--

```
/*************************************************************************
 *      ChessMovs.e -- Written in ClassC by Paul Kirkaas --
 *
 *      ChessMovs shows all legal chess moves for any given
 *      chess board configuration.
 *      The chess board and each piece are objects.  The pieces
 *      examine the board and the other pieces on it, and generate
 *      their own legal moves based on their information about themselves
 *      and the state of the board.
 *************************************************************************/

#include "List.h"
class Board
{  Piece square[8][8];  // array of Pieces
   display();
   setup();
   char  sect[8][8][9];
   object moves; // Set of moves -- gets set/initialized by the piece
   void add();
   delete();
};
class Pos { int x;
          int y;
          char * eval();
          object copy();
          };
class Piece
{
  object pos; /* position object, of class Pos */
  char colr;
  void showmoves();
  object genmoves(); /* Returns an object of class Set, of moves */
  char* eval();
  delete();
  init();
  char id();
 };
object Piece::genmoves() {}
char Piece::id() {}
class Rook : Piece
{
 char id();
 object genmoves(); /* Returns an object of class Set, of moves */
 object Vmoves();
 };
class Bishop : Piece
{char id();
```

```
 object genmoves();
 object DiagonalMoves();
 };


class Queen : Rook : Bishop

// An example of multiple inheritence -- the queen has characteristics of
// both the rook and bishop.

{char id() ;
 object genmoves();
 };

extern Board board;
Board::delete()
{ int i,j;
  for ( i=0; i < 8; i++) for ( j=0; j < 8; j++) square[i][j]:>delete();
  moves:>delete();
}


Board::init()
  {
   int i,j;
   for (i=0; i<8; i++) for (j=0;j<8;j++)square[i][j] = 0;
  }
void Board::add(pP, i, j) Piece pP; int i,j;
{square[i][j] = pP;}
Board::display()
{ setup();
  char line[99];
  strcpy(line,
       "_____________________________________________________________________________________0);
  printf("[H%s",line); // Go Home & draw 1st line.
  int i,j;
  for (j=0;j<8;j++)
    {
    for (i=0;i<8;i++) printf("|%s",sect[i][7-j]);
    printf("|80s",line);
    }
};

Board::setup()
 { int i,j;
   for(i=0;i<8;i++) for (j=0;j<8;j++)
     { strcpy (sect[i][j],"      ");
       if (square[i][j])
         {
          sect[i][j][1] = square[i][j]:>colr;
          sect[i][j][3] = (square[i][j]):>id();
         }
     }
  if (moves)
  {
```

```
    moves:>rewind();
    while (moves:>place)
      { sect[moves:>current():>(int)x][moves:>current():>(int)y][5] = 'X';
       moves:>next();
      }
    }
  };

/*************************************************************************/

char * Pos::eval()
{static char  PosEvalRet[2][99];
 static int ix = 0;
 ix = !ix;
 sprintf(PosEvalRet[ix],"Pos:x=%d; y=%d :: ",x,y);
 return PosEvalRet[ix];
}
object Pos::copy()
{ object retobj = Pos:>new();
  retobj:>(int)x = x;
  retobj:>(int)y = y;
  return retobj;
}
Piece::delete() { pos :> delete(); }
Piece::init(cl,x,y)char cl;int x; int y ;
  {
   pos = Pos:>new();
   colr = cl;
   extern object board;
   board:>add(self,x,y);
   pos:>(int)x=x;
   pos:>(int)y=y;
}

char* Piece:: eval()
   {char outS [2][99];
    static int ix = 0;
    ix = !ix;
    sprintf(outS[ix]," Piece:: %c [%d,%d]0,colr,
            pos:>(int)x,pos:>(int)y);
    return outS[ix];
   }

void Piece::showmoves(){// HERE IS WHERE WE NEED GARBAGE COLLECTION
                board :> moves  =genmoves();
                board :> display();
               };

char Rook::id() {return 'R';};

object Rook::genmoves()
{ return Vmoves();}

object Rook::Vmoves()
{ Pos tstpos;
```

```
 object pieceP=0;
 object moves = Set:>new();
 tstpos = pos:>copy();

// Rook movement in Positive x direction.

 for ( (tstpos:>x) = pos:>(int)x+1;
       (tstpos:>x) < 8; (tstpos:>x)++)
    { pieceP =(board:>square[tstpos:>x][tstpos:>y]);
     if (pieceP)
       { if (pieceP:>(char)colr != colr) moves:>add(tstpos:>copy());
         break;
       }
     moves:>add(tstpos:>copy());
    }

tstpos = pos:>copy();
// Rook movement in Negative x direction.

 for ( (tstpos:>x) = pos:>(int)x-1;
       (tstpos:>x) >= 0; (tstpos:>x)--)
    { pieceP =(board:>square[tstpos:>x][tstpos:>y]);
     if (pieceP)
       { if (pieceP:>(char)colr != colr) moves:>add(tstpos:>copy());
         break;
       }
     moves:>add(tstpos:>copy());
    }


// Rook movement in Positive Y direction.

tstpos = pos:>copy();
 for ( (tstpos:>y) = pos:>(int)y+1;
       (tstpos:>y) < 8; (tstpos:>y)++)
    { pieceP =(board:>square[tstpos:>x][tstpos:>y]);
     if (pieceP)
       { if (pieceP:>(char)colr != colr) moves:>add(tstpos:>copy());
         break;
       }
     moves:>add(tstpos:>copy());
    }

// Rook movement in Negative Y direction.
tstpos = pos:>copy();

 for ( (tstpos:>y) = pos:>(int)y-1;
       (tstpos:>y) >= 0; (tstpos:>y)--)
    { pieceP =(board:>square[tstpos:>x][tstpos:>y]);
     if (pieceP)
       { if (pieceP:>(char)colr != colr) moves:>add(tstpos:>copy());
         break;
       }
     moves:>add(tstpos:>copy());
    }
```

```
return moves;
}

/*************************************************************************/

object Bishop::genmoves()
{ return DiagonalMoves(); }

char Bishop::id() {return 'B';};

/*************************************************************************/

object Bishop::DiagonalMoves()
{ object tstpos;
  object pieceP;
  object moves = Set:>new();
  tstpos = pos:>copy();
/* Generate diagonal move for Bishop in Positive X & Positive Y */
  for ( (tstpos:>(int)x) = pos:>(int)x+1,(tstpos:>(int)y) = pos:>(int)y+1;
        ((tstpos:>(int)x) < 8) && ((tstpos:>(int)y) < 8);
        (tstpos:>(int)x)++, (tstpos:>(int)y)++)
    { pieceP =  board:>square[tstpos:>x][tstpos:>y];
      if (pieceP)
        { if (pieceP:>(char)colr != colr)  moves:>add(tstpos:>copy());
          break;
        }
      moves:>add(tstpos:>copy());
    }


tstpos = pos:>copy();
/* Generate diagonal move for Bishop in Negative X & Negative Y */

  for ( (tstpos:>(int)x) = pos:>(int)x-1,(tstpos:>(int)y) = pos:>(int)y-1;
        ((tstpos:>(int)x) >= 0) && ((tstpos:>(int)y) >= 0);
        (tstpos:>(int)x)-, (tstpos:>(int)y)-)
    { pieceP =  (board:>square[tstpos:>(int)x][tstpos:>(int)y]);
      if (pieceP)
        { if (pieceP:>(char)colr != colr)  moves:>add(tstpos:>copy());
          break;
        }
      moves:>add(tstpos:>copy());
    }


/* Generate diagonal move for Bishop in Negative X & Positive Y */
tstpos = pos:>copy();
  for ( (tstpos:>(int)x) = pos:>(int)x-1,(tstpos:>(int)y) = pos:>(int)y+1;
        ((tstpos:>(int)x) >= 0) && ((tstpos:>(int)y) < 8);
        (tstpos:>(int)x)-, (tstpos:>(int)y)++)
    { pieceP =  (board:>square[tstpos:>(int)x][tstpos:>(int)y]);
      if (pieceP)
        { if (pieceP:>(char)colr != colr)  moves:>add(tstpos:>copy());
          break;
        }
```

```
    moves:>add(tstpos:>copy());
   }


/* Generate diagonal move for Bishop in Positive X & Negative Y */
tstpos = pos:>copy();
 for ( (tstpos:>(int)x) = pos:>(int)x+1,(tstpos:>(int)y) =
          pos:>(int)y-1;
      ((tstpos:>(int)x) < 8) && ((tstpos:>(int)y) >= 0);
      (tstpos:>(int)x)++, (tstpos:>(int)y)--)
   { pieceP =  (board:>square[tstpos:>(int)x][tstpos:>(int)y]);
     if (pieceP)
       { if (pieceP:>(char)colr != colr)  moves:>add(tstpos:>copy());
         break;
       }
     moves:>add(tstpos:>copy());
   }
return moves;
};



/*************************************************************************/
char Queen::id() {return 'Q';}

object Queen::genmoves()
{ object Rmoves;
 object Bmoves;
 Rmoves = Vmoves();
 Bmoves = DiagonalMoves();
 Rmoves:>merge(Bmoves);
 return Rmoves;
}
/*************************************************************************/
Board board;

main()
{
   free(malloc (1000000));
   system("clear");
   board = Board:>new();
   // Place a piece on the board by declaring it and positioning it as follows:
   object WhiteRook = Rook:>new ('w',4,5);
   object BlackRook = Rook:>new('b',5,5);
   object WhiteBishop = Bishop:>new ('w',2,3);
   object WhiteQueen = Queen:>new ('w',2,5);
   /*
   int i;
   for (i=0; i<100; i++)
   */
   while(1)
   {
      // Each piece will generate its legal moves if so requested:
      WhiteRook:>showmoves();
      WhiteQueen:>showmoves();
```

```
        // WhiteBishop:>showmoves();
    }
}
```

```
/*************************************************************************
 *      List.e -- Written in ClassC by Paul Kirkaas --
 *
 *      "List.e" creates and defines the abstract data type "list"
 *      for use in the ChessMovs.e program
 *************************************************************************/


#include "List.h"

eqv(a,b) object a,b; /* Compares the eval() strings of the two objects
                      * and returns 1 if they are equivalent; else 0 */
{ return !(strcmp (a:>eval(), b:>(char *) eval())); }

char * ListElement::eval() { return self:>entry:>(char *)eval(); }

List::delete()
{
   head:> delete();
   place:>delete();
}

ListElement::delete()
{
   nextElement:>delete();
   entry:>delete();
}


void Set::add(obj) object obj;
{ self:>rewind();
  object temp;
  while (temp = current())
    {
      if (eqv(temp, obj)) return;
      next();
    }
  ListElement newel = ListElement:>new ();
  newel:>entry = obj;
  newel:> nextElement = self:>head;
  head = newel;
}

void Set::remove(obj) object obj;
{
  object x;
  rewind();
  while (place)
  { if (eqv (current(),obj)) // The current entry does match obj
      { if (x) // If the previous link is !0;ie, if self is not first element
          { x:>nextElement = place:>nextElement;
            place = x:>nextElement;
          }
        else  // The object was the FIRST element of the list
          { head = place:>nextElement;
```

```
            place = head;
          }
        }
      else  // The current entry does not match obj -
        { x =  place;
          next();
        }

    }
}
void Set::merge(set) object set; /* merge two sets */
{ set:>rewind();
  object temp;
  while ( temp = set:>current() )
       { add(temp);
         set:>next();
       }
}

int List::init()
{ head = 0;
  place = 0;
}


object List::current()
{ if ( place) return place:>entry;
  return 0;
}
void List::next()
{ if (place)
    place = place:>nextElement; }

void List::rewind() {place =  head; }

void List::add(obj) object obj;
{ ListElement newel = ListElement:>new ();
  newel:>entry = obj;
  newel:> nextElement = head;
  head = newel;
}

void List::remove(obj) object obj;
{
 object x;
 rewind();
 while (place)
 {  if (  place :> entry == obj) // The current entry does match obj
      { if (x) // If the previous link is !0;ie, if self is not first element
          { x:>nextElement =  place:>nextElement;
            place = x:>nextElement;
          }
        else  // The object was the FIRST element of the list
          {  head =  place:>nextElement;
             place =  head;
```

```
        }
      }
    else  // The current entry does not match obj --
      { x = place;
        next();
      }

  }
}
```

```
/*********************************************************************
 *      List.h -- Written in ClassC by Paul Kirkaas --
 *
 *      List.h contains the declarations for the class "list"
 *      which is used in "ChessMovs" and defined in "List.e"
 *********************************************************************/


class ListElement
{ object entry;
  ListElement nextElement;
  delete();
  char * eval ();
};

class List
{ ListElement head;
  ListElement place;
  object current();
  void next();
  void rewind();
  void add();
  void remove();
  int init();
  delete();
};

class Set : List
{ void add();       /* Only adds if equiv object not already in set */
  void remove(); /* Compares by eval string (eqv()); not object # */
  void merge();  /* Combines two sets in one */
};
```

5.2 Backprop.e--

```
/************************************************************************
 *      Backprop.e -- written in ClassC by Paul Kirkaas
 *      Neural Network simulation program:
 *      Learns exclusive OR through trial and error.  Output reports the
 *      percentage error after each set of one hundred test cases.
 ************************************************************************/
#include <stdio.h>
#define NO_RUNS 100000
#define NO_PER_SET 100
#define outfile "backprop.rslt"
double exp();
FILE * outFP;
double c = .5;
class Neuron
{
  Neuron to[10];
  Neuron from[10];
  double w[10];
  double q;
  double delta;
  init();
  forward();  /* Applies weights to inputs & generates output */
  back();     /* Examines its given deltas & sends back */
};

double drand48();

int cycle = 0;
main()
{
  Neuron A0 = Neuron:>new();
  Neuron A1 = Neuron:>new();
  Neuron A2 = Neuron:>new();
  Neuron A3 = Neuron:>new();
  Neuron A4 = Neuron:>new();
  Neuron A5 = Neuron:>new();
  connect(A0,A3);
  connect(A0,A4);
  connect(A0,A5);
  connect(A1,A3);
  connect(A1,A4);
  connect(A2,A3);
  connect(A2,A4);
  connect(A3,A5);
  connect(A4,A5);
  srand48(time(0) * getpid());
  int test_pattern;
  for(cycle =0; cycle < NO_RUNS; cycle++)
  {
    test_pattern = input(A0,A1,A2);
    A3:>forward();
    A4:>forward();
```

```
    A5:>forward();
    analyze(test_pattern,A5);
    A5:>back();
    A4:>back();
    A3:>back();
  }
}

analyze(pat,nron) int pat; Neuron nron;
{int des =  ((pat & 1) || (pat & 2)) && (!((pat & 1) && (pat & 2))) ;
 double result = nron:>q;
 double error = (des ? (.9 - (double)result) : ((double)result - .1));
 if (error < 0.) error = 0.;
 static double cum_err = 0;
 cum_err += error;
 if (! (cycle % NO_PER_SET))
   {
    printf("%d: %f %% error0,cycle,cum_err);
    cum_err = 0;
   }
 nron:>delta += (des - result)* result * (1 - result);
 c = 2*error + .3;
}

 connect(obA,obB) Neuron obA,obB;
 { int i = 0;
   for (i=0; i<10; i++)
     { if (! (obA:> to[i]))
         { obA:> to[i] = obB;
           break;
         }
     }

   for (i=0; i<10; i++)
     { if (! (obB:> from[i]))
         { obB:> from[i] = obA;
           break;
         }
     }
 }

 Neuron::init()
 { int i;
   for (i = 0 ; i < 10; i ++) w[i] = drand48()-.5;
 }

 double f(net) double net;
 { return 1/(1 + exp( -net));}
 Neuron::forward()
 // Examines its input, decides on its output
 {
  int i;
  double q = 0;
  double net = 0;
  for (i = 0; i < 10; i ++)
```

```
    {if (! from[i]) break;
     net += w[i] * from[i]:>q;
    }
 q = f(net);
}

Neuron::back()
// Computes its weight changes and the delta for the previous units
{
 int i;
 double my_delta = delta;
 delta = 0;
 double her_q = 0;
 double my_q = q;
 for (i = 0; i < 10; i ++)
    { if (! (from[i])) break;
      her_q = from[i] :> q;
      /* First calculate her_delta, becuase it is based on CURRENT weights*/
      /* Use "+=" for the delta, because in the general case, additive */
      from[i]:>delta += my_delta*w[i]*her_q*(1-her_q);
      /* Now calculate weight changes */
      w[i] += c*my_delta*her_q; }
}
input (ob0,ob1,ob2) Neuron ob0,ob1,ob2;
{
  int pat = 4 * drand48();
  ob0:>q = (double)1;
  ob1:>q = (double) (!!(pat&1));
  ob2:>q = (double) (!!(pat&2));
  return pat;
}
```