View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by NASA Technical Reports Server

IN-61

31430

26P

NASA Technical Memorandum 106315

Flexible Method for Inter-Object Communication in C++

Brian P. Curlett and Jack J. Gould
Lewis Research Center
Cleveland, Ohio

November 1994

National Aeronautics and
Space Administration

(NASA-TM-106315) FLEXIBLE METHOD FOR INTER-OBJECT COMMUNICATION IN C++ (NASA. Lewis Research Center) 26 p

N95-17420

Unclas

G3/61 0031430

Flexible Method for Inter-Object Communication in C++

Brian P. Curlett and Jack J. Gould

National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135

Summary

A method has been developed for organizing and sharing large amounts of information between objects in C++ code. This method uses a set of object classes to define variables and group them into tables. The variable tables presented here provide a convenient way of defining and cataloging data, as well as a user-friendly input/output system, a standardized set of access functions, mechanisms for ensuring data integrity, methods for interprocessor data transfer, and an interpretive language for programming relationships between parameters.

The object-oriented nature of these variable tables enables the use of multiple data types, each with unique attributes and behavior. Because each variable provides its own access methods, redundant table lookup functions can be bypassed, thus decreasing access times while maintaining data integrity. In addition, a method for automatic reference counting was developed to manage memory safely.

1 Introduction

Object-oriented programming techniques solve many of the problems involved with data organization in large software systems. These techniques accomplish this by encapsulating the data, along with the methods of solution, for a particular piece of the problem. However, difficulties still arise when objects have to share the same data. This also poses the problem that each object must provide its own input/output (I/O) methods and its own data validation methods. This report presents our solution to these problems.

Our approach uses a set of C++ classes that define and organize the attributes of an object. In addition to the value of the attribute, these classes can store other relevant information, such as a description of the attribute, the default value, and limits. They can also group attributes into tables. We refer to these attribute classes as *variables* and to the tables as *variable tables* or *data dictionaries*. The variable tables provide a convenient way to define and catalog data, as well as a user friendly I/O system, a standardized set of access functions, mechanisms for ensuring data integrity, and an interpretive language for programming relationships between parameters.

Using variable tables as a compromise between a full-blown data base management system and simple global data structures is not a new idea. Variable tables based on the same basic principles as this one have been used successfully for engineering applications by others. Kroo [1] used a variable table for an aircraft design system written in FORTRAN. He indicates that this method of data storage greatly enhanced the extensibility and maintainability of his system. Curlett [2] used a variable table programmed in C for graphical user interface applications, also with success.

A variable table written in C or FORTRAN maintains data integrity by prohibiting any direct access to the data in the table. Instead, the programmer must use *set* and *get* functions to store or retrieve data, respectively. These functions pass a key (usually a string representing the variable name) with each set or get call. The table must be searched for the key before the data can be manipulated.

With the object-oriented variable table presented here, the programmer does not need to search the table every time the program needs to access a variable. Instead, the programmer looks up the variable object once, stores its address, then accesses it directly without going through the table. Data integrity is maintained because the variable object has its own set and get functions.

There are several other advantages to an object-oriented variable table. One is that new variable types can be derived easily by using inheritance from existing types. Another is that a variable table may itself be a variable, and therefore, be stored within another table. This permits hierarchical variable tables to be constructed as a tree, similar to the way a DOS or UNIX file system is constructed. In addition, the function and operator overloading capabilities in C++ make for a robust interface to the variables and tables.

The use of this object-oriented variable table can save the programmer many hours by eliminating much of the mundane work involved in programming data structures and data I/O routines. Use of this table can also save the program user many hours because of its built-in data validation, easy-to-read file formats, and programmability.

This report explains the variable table concept and the features of this implementation in detail. The reference manual that is included with the source code should be referred to for programming details. Anyone programming with our variable table code should be familiar with this report. This report will also be of interest to C++ programmers looking for alternative data-handling techniques. We assume a basic understanding of object-oriented programming terminology and the C++ programming language [3].

This report first gives an overview of variable tables. Section 3 then discusses the class hierarchy of the variables. Sections 4 and 5 describe the file formats for defining variables and entering values, respectively. Section 6 describes how to use these classes in a C++ program, and section 7 describes a template class that links existing data types into the variable table. Some ideas for adding time history storage are presented in section 8, and section 9 explains how the built-in programming language works. Sections 10 and 11 explain some additional functionality in the variable classes for supporting graphical user interfaces and distributed processing, respectively. Finally, section 12 discusses how to use reference counting for memory management.

2 Overview

Before discussing the class structure and the programming details, we first look at how these variable tables are used in an object-oriented program. We use variable tables for objects that represent major components of a program, objects that need to interact with the user through an interface, or objects that have many I/O parameters. This code was developed primarily as part of a large simulation framework. The major objects in this framework are referred to as *components.* In this report, the term *component* refers to an object that uses the variable table.

When a component is created, it loads a file that defines the variables in its table. This file is called the data definition file. (See section 4 for details; fig. 2 shows an example.) The data definition file contains information such as variable type, variable name, minimum and maximum allowable values of the variable, default value, help information, and a label string. We have defined eight types of variables that can be created: basic, numeric, option, Boolean, string, file, link, and

table. The attributes in the definition file depend on the type of variable. A plain ASCII text definition file stores this information in a convenient format that allows users to modify it quickly without recompiling the code.

Because there may be several instances of a given component type, it is desirable to store the values and definitions of the variables in separate files. Functions are provided to read these files of "instance" values after the definition files have been loaded. These files are referred to as ***input files***. Section 5 explains the use of I/O files, and figure 3 shows what one of these files may look like.

After the component loads its table and reads its instance values, it can search the table for variables by using the variable name as the search key. The search function returns a pointer to the variable object. This pointer can be stored in the component to avoid subsequent searching. Because the variable itself is an object and provides safe access mechanisms to the data within (there are no public data in a variable object), the storage of this object pointer does not pose a problem with maintaining data integrity.

The component can now use set and get functions to store and retrieve data to and from the variables. These functions take on slightly different forms depending on the type of variable, but the syntax is kept as similar as possible to avoid confusion. For convenience, some types overload the set function and provide multiple get functions to retrieve data in different formats. For example, a numeric variable can be set by passing it a number or an equation as a character string. Some operators have also been overloaded for convenience. Section 6 explains how to use lookup, set, and get functions in a program.

So far, we have described only functionality for cataloging the attributes of a component, but our major objective is to provide a mechanism for sharing data between components. This is done by placing each component's variable table into a global variable table. The system and other components can then access any data by specifying the component's table name followed by a backslash (\) and then the attribute's name while searching the main variable table.

Next we consider a graphical user interface for an engineering application. In this application, two objects need to share the same data. One object represents the analysis of a part, and the other object displays information about that part on the screen. By using a variable table, we can program the two objects independently from one another, but the objects can still interact. *Callbacks*, functions called when a variable's value is changed, are added to the variable by each object. So when one object changes the variable's value, the other object can update itself. With this method, the analysis portion of the code is completely oblivious to the user interface; it needs only to save its data into its variable table, and the user interface will update its display automatically. The separation of the user interface and analysis programming tasks greatly simplifies the development of a large simulation program. These same mechanisms for data communication are used between two analysis components and between analysis components and the system solver. Section 10 discusses further the use of variable tables in conjunction with a graphical user interface.

3 Class Hierarchy

All variables, including variable tables, are derived from the Var class (fig. 1). The Var class holds information common to all variables, such as name, label, and help information. Because all variables can be declared as arrays, Var also stores array dimensions. The Var class provides access functions to all these data.

The VarNumeric class stores information for numeric data types. There is no distinction between integer and floating point data in this class. In addition to the value of the variable, the VarNumeric

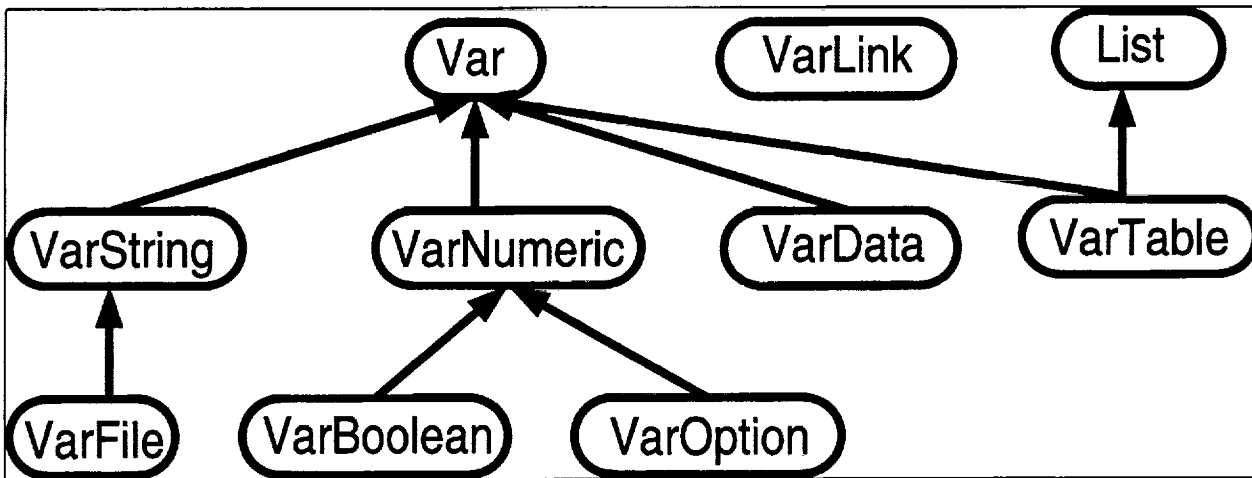

Figure 1: Variable table class structure.

class stores the default value, minimum allowable value, and maximum allowable value. All four of these parameters are stored as one-dimensional arrays of "smart" pointers to objects of the Eq class, which are dynamically allocated as needed by the VarNumeric class (see section 9). The Eq class stores and evaluates equations. It can store a character representation of the datum, a pointer to the parse tree of the string, and a double-precision floating point number. The tree is stored so that the character string does not need to be re-parsed each time the equation needs to be evaluated. If a floating point number is entered instead of an equation, no parse tree is needed, and therefore, it is not constructed. The Eq class is completely in line and, therefore, does not slow the execution of the VarNumeric class.

The VarBoolean class is derived from the VarNumeric class for handling Boolean data types. Boolean is normally a True/False (1/0) selection, but it can be used to select between any two values. VarBoolean uses the same storage as VarNumeric; a value equal to the minimum is considered False, and a value equal to the maximum is considered True.

An option, or enumerated, data type is also derived from the VarNumeric class. This class is called VarOption. The VarOption class allows the user to choose one value from a list of options. A class for handling character string data is also provided. This class, which is derived from the Var class, is called VarString. VarString stores an array of pointers to character strings. It also stores default values.

VarFile, a class for handling files, is derived from VarString. VarFile primarily distinguishes between file names and other strings in a graphical user interface. The VarFile class also stores attributes for the directory name and the file search mask.

All the variable classes mentioned previously store their values internally. There are cases where the user will want to access the values of an object either directly or through member functions. Two parameterized classes provide this functionality: `VarLink<ClassName,TypeName>` and `VarData<TypeName>`. Using these classes eliminates the need for updating the variable table when the object's value changes. VarLink allows the user to set and get through member functions. This is very helpful for cases where a member function causes side effects in the object when it sets or gets a value or when the value is not stored but derived from other values.

VarTable is a container class that holds other variables. This class inherits from both the List container class and the Var base class. Because the VarTable class is derived from the Var class, a VarTable object is itself a Var and can, therefore, be placed in other VarTable objects.

4 Data Definition File

Although variables and tables can be created programmatically, we recommend that they be loaded from a file. This section explains how to create files that define variable tables. These files are called *data definition files.*

As stated previously, there are eight types of variables that can be entered through a definition file. They are

Var — basic variable, holds only the label and help information

VarString — holds character string data

VarFile — holds file names

VarNumeric — holds numerical data

VarBoolean — holds Boolean (True/False) data

VarOption — holds enumerated data

VarLink — links a variable to data stored elsewhere

VarTable — holds other variables

All the data types use basically the same input syntax; however, attributes for each type differ. See the reference manual that is included with the source code for a list of attributes that can be defined for each variable type in the definition file.

The grammar for defining a variable is shown in appendix A. The input is similar to that for declaring structures in the C programming language. A variable or variable table is introduced by a *type-name* followed by an *identifier.* The keyword `typedef` can prefix a variable table definition to introduce the name of the new variable table as a type-name for declaring subsequent variable tables.

Following the definition of the variable, the variable's attributes can be entered in a statement block surrounded by braces. The name of the attribute is followed by an equals sign, then the value of the attribute. The order in which the attributes are entered does not matter.

Some attributes can be entered as one- or two-dimensional arrays. In this case, the elements of the array are separated by commas. Two-dimensional arrays are indexed by row first as in the C programming language (in FORTRAN the indexing is reversed). Subscripts also can be used to indicate the position in an array. Array subscripts start at 0 (not 1).

```
VarNumeric x[20] {
  max[5] = 5;
  max[10] = 10;
  max[15]=15, 16, 17, 18, 19;
}
```

Character string attributes should be placed in double or single quotation marks. For example,

```
VarString names[2] {
  default = "apples, pears, grapes", "beef, pork";
}
```

Without quotation marks, the string would have been split at the first comma after "apples."

White space in the definition file has no significance. Also, C and C++ style comments can be used before or after any input line in the file. However, comments are not valid in the middle of an expression.

Multiple instances of the same variable definition can be created by placing an *identifier list* and a semicolon after the variable definition. For example,

```
VarNumeric xx {
  min=0; max=10;
  label = 'label line for xx, yy, and zz';
} yy, zz;
```

the variables **xx**, **yy**, and **zz** all share the same definition.

A definition of a variable table begins as for any other variable. All variables within the definition of the table are considered part of that table. Unlike variables, a variable table definition must be followed by a semicolon. The semicolon can be preceded by an optional identifier list, which names variable tables that are copies of the variable table just defined. An example of a simple data definition file is shown in figure 2.

```
// C and C++ style comments may be used
VarTable Input {
  label = "label line for VarTable Input";
  help  = "help info for VarTable Input";

  VarString name {
    Label = "Enter a name for this component:"
    default = "Turbine";
  }

  VarNumeric YY {
    label = "Enter one-line label string here";
    help = "Multiple lines of help information
            may be entered here.";
    default = 10;
  }

  VarNumeric XX[2][3] {
    label = "Enter one-line label string here";
    default = 0, YY, 2*YY, 3*YY, 4*YY, 5*YY;
    min     = 0,0,0,0,0,0;
    max     = 100,200,300,400,500;  // max[1][2] is undefined
  }
} Inputs_2, Inputs_3; // end of VarTable Input
```

Figure 2: Sample input data definition file.

The example constructs a variable table that has three variables. The first variable is a string, the second is a numeric variable, and the third is a two-dimensional array of numeric variables.

Two copies of this table are made and given the names Inputs_2 and Inputs_3.

Sometimes it is desirable to include one definition file in another. If, for example, several components all share some common subcomponent, then the definition file for each component can include the same definition file for the subcomponent. This is done by using an #include statement.

```
VarTable T1 {
  #include "input_port.def"
  #include "output_port.def"
  #include "mech_port.def"
}; // end of T1
```

The contents of the three files specified are included in the definition file for T1. The only requirement for using "include" files is that the resulting input, when read in the specified order, be syntactically correct. There is no restriction on the nesting of files or on the resulting input size. The path for locating the include files is the same as the path for locating the definition file.

5 Input/Output Files

This section describes the I/O syntax used for variables and variable tables. Since I/O functions are called recursively for subtables, the easiest method for handling all input and output to a program is to place all tables in one main variable table and read and write that table from and to an I/O stream.

Figure 3 shows a simple I/O file.

```
main {
  T1 {
    AA=3.5;
    XX=1,2,3;
    YY=2*XX[1];
  }
  T2 {
    ZZ=\T1\AA;
  }
}
```

Figure 3: Sample input/output file for a variable table.

The input file is free format. White space is ignored, and an input line ends with a semicolon. A table is started with the table name followed by a left brace, and the table ends with a corresponding right brace. This ensures proper nesting of the tables. All input data within the braces are part of that table. It is helpful to include C++ style comments at the end of a table. This is done automatically if the table is output in this format.

The order in which variables occur in a table does not matter. Arrays are treated the same as in the definition file. It is not a requirement to input all variables in a table. However, it is erroneous to specify a variable in a table that does not exist in that table.

Note that the variable ZZ is programmed to be equal to the value of variable AA in table T1. The backslash (\) in front of the table name means "start the search for that variable in the main

variable table." That is, the string "\T1\AA" means "find the table T1 in the main variable table, then search that table for the variable AA."

An alternative method to grouping all variables in nested tables is to specify each variable name with the table name and a back slash(\) preceding it. For example,

```
main {
  T1\AA=3.5;
  T1\XX=1,2,3;
  T1\YY=2*XX[1];
  T2\ZZ=\T1\AA;
}
```

is equivalent to the input file shown in figure 3.

C and C++ style comments can be used in the input file with the restrictions mentioned in section 4.

If a label string exists for a given variable, the output file includes this label as a comment after the variable and value.

```
T1 {                    // Label for table T1 is placed here
  AA=3.5;               // Label line for AA
  XX=1,2,3;             // Label line for array XX
}
```

Although the labeling information stored with each variable is intended primarily for the user interface, it does make these I/O files easily readable and is quite useful for batch-oriented programs. The labels in an older input file can be updated to the labels in a newer definition file by running a simple program that reads the table and writes it back out. Such a program is presented in section 6 to explain the basics of using variable tables in a C++ program.

6 Programming with Variable Tables

This section discusses some of the most frequently used methods. Details on all the methods available are given in the reference manual that is included with the source code.

Every program using variable tables must have at least two tables constructed: MainVarTable (the top level table) and calcVarTable (the table used for temporary variables by the calculator, section 9). The Var::Initialize() static member function creates these two tables for the user and does all the other necessary setup. The main program should start like this:

```
// this includes all the Var class header files
#include <VarClasses.H>
main()
{
   Var::Initialize();
   // ...
}
```

6.1 Loading a Definition File

After MainVarTable and CalcVarTable are created, other variables are created by loading them from definition files. The function load() is used to load a variable from a stream. The user has the option to read the entire file, which is the default, or to read one variable at a time.

The user also has the option of using a global parser or a parser that is local to the variable table. The global parser is the default and is the most useful. The local parser option is useful if the user wants to read values for multiple variable tables asynchronously. `load()` is a member function of `VarTable` and can be used as shown below:

```
ifstream strm("myTable.def");
VarTable *myTable = new VarTable("myTable");
Var *new_var;

// read entire file using global parser...
myTable->load(strm);

// or
strm >> myTable;

// read entire file using local parser...
myTable->load(strm, readAllVariables, useLocalScanner);

// read first complete variable using global parser...
myTable->load(strm, readSingleVariable);

// read first complete variable using local parser...
myTable->load(strm, readSingleVariable, useLocalScanner);
```

The `load()` function returns a pointer to the last variable read from the input stream. If a table is read, then a pointer to the table is returned—not a pointer to the last variable in the table.

The programmer should check the existence and type of Var returned from `load()` before attempting to cast it to `VarTable`.

```
Var *tmp = myTable->load(cin);
if (tmp == NULL || !tmp->isA(VarTableClass)) {
   cerr << "Invalid input\n";
   exit(1);
}
VarTable *tab = (VarTable *) tmp;
```

The `isA()` method will return True if the object `*tmp` is of the VarTable class or of a class derived from VarTable. `VarTableClass` is part of an enumeration that identifies all the classes. The `classType()` method returns this enumerated value. The `if` statement just given could have been written as

```
if (tmp == NULL || tmp->classType() != VarTableClass) { //... }
```

However, if a new type of table class was derived from the VarTable class, the latter method would produce an error. Which of these two methods is more appropriate to use will depend on the circumstances.

6.2 Input/Output Methods

Each variable has methods for writing its definition and values to an output stream. Information is input solely through the input parser, which can differentiate between variable definitions and

variable assignments. To make variable table output convenient, there are three output manipulators, which include `varFormat()`, `varDef()`, and `varInp()`. Each of these has a pointer to a variable table as its sole argument. `varDef()` writes the variable table definition (e.g., name, label, help, min, max, etc.) to the output stream. `varInp()` writes the values (e.g., xx = 1.1234;) to the output stream. `varFormat()` writes `varDef()` followed by `varInp()` to the output stream. The following two statements are equivalent:

```
cout << varDef(myTable) << varInp(myTable);

cout << varFormat(myTable);
```

We are now ready to write a program for reformatting an input file.

```
#include <VarClasses.H>
int main(int argc, char** argv)
{
  Var::Initialize();
  MainVarTable->load(argv[1]);          // load the definition file
  cin  >> *MainVarTable;                // load the instance values
  cout << varInp(MainVarTable);         // write out the instance values

  return 0;
}
```

This program loads a variable table or tables from the file specified on the command line. It reads input values from standard input and then it writes all the values back out (nicely) to standard output. Of course, a real program should test for the existence of the definition file and make sure that the table was correctly loaded.

6.3 Locating Variables

As just illustrated, variables are usually created by loading variable tables from data definition files. A particular variable in a table can be accessed by using the `lookup()` method.

```
Var *tmp = tab->lookup("xx");
if (tmp == NULL || tmp->classType() != VarNumericClass) {
     // handle error
}
VarNumeric *v_xx = (VarNumeric *) tmp;
```

Because `lookup()` is often used just to check the existence of a variable, it does not produce any type of error message; if it cannot find a variable, it returns NULL. It is critical that the programmer check the returned value for existence and type before using it!

Convenience routines are provided to simplify the error checking and type casting; these are `lookupString`, `lookupFile`, `lookupNumeric`, `lookupOption`, `lookupBoolean`, and `lookupTable`. These functions will print an error message if a variable of the correct type is not found. They return a pointer cast to the correct type. For example, the previous code fragment could have been written as

```
VarNumeric *v_xx = tab->lookupNumeric("xx");
if (v_xx == NULL) {  // handle error }
```

Although these convenience functions will print a warning message if the lookup fails, they will not stop the execution. Until more sophisticated error handling becomes available for our framework, the calling routine is responsible for trapping the error.

Note that it is not necessary to specify `MainVarTable` when searching the main variable table. For example, `lookupNumeric(name)` is the same as `MainVarTable->lookupNumeric(name)`.

To avoid redundant lookups, it is usually best that each major component in a simulation program have its own variable table. This table is loaded from the file during the construction of that component. Variable pointers are then looked up from the table during the construction of the component.

Sometimes it is desirable to operate on all the values in a table without having to use `lookup()` for each variable. This is done with the [] operator. The [] operator takes an array index as an argument and returns a pointer to the variable at that location in the table. For example,

```
for (int i=0; i < tab->size(); i++)
   cout << (*tab)[i]->getName() << endl;
```

prints the name of each variable in the VarTable `*tab` to standard output.

6.4 Accessing Values

When to retrieve data from a variable depends on the use of the variable. Some variables may be constant. In this case, their value needs to be retrieved only once during a component's construction. The values of some variables may be modified by other objects. In this case, the value of the variable may have to be retrieved each time it is used in the component. In other cases, a variable is used for data output only; therefore, its value may never need to be retrieved by the component.

Functions are provided to retrieve all attributes of each Var class. These are discussed in the reference manual. Some attributes can be retrieved in more than one format. For example, the VarNumeric class provides functions to retrieve a value as `double` or as `char *` (the latter is important for user interface programming). Because C++ does not support overloading on the basis of the returned value of a function, different names had to be given to functions with different return data types. However, set functions are named the same regardless of the type of data being passed to the object.

The following code fragments will help clarify how variables are used in a program.

```
VarNumeric *var = lookupNumeric ("\\T1\\CC");
if (var == NULL) {
   /* handle error */
}
double cc = var->get();  // returns first numerical value in \T1\CC
cc = var->get(1);        // returns second numerical value in \T1\CC
cc = (*var)[1];          // same as preceding fragment w/ different syntax

double *arr = var->getArray(2,3);  // returns an array of 3 values
                                   // starting at index 2
char **str = var->getStrings(2,3); // returns an array of 3 strings
                                   // starting at index 2

var->set(10.0);          // sets the first value to 10.0
var->set(20.0,1);        // sets the 2nd value in the array to 20.0
var->set("10.0,20.0");   // sets 1st and 2nd values using a string
```

The [] operator for the VarBoolean class returns True or False. For the VarString and VarFile class, this operator returns `const char*`.

6.5 Callbacks

A *callback* is a procedure registered with a variable object. The callback is invoked as a side effect from an operation on the variable. For example, the `VarValueChangedCallback` is invoked when the value of the variable is modified, and the `VarDestroyedCallback` is invoked just before the variable is deleted.

Callbacks are added by using the `addCallback(type, CB, clientData)` member function. The first argument (`type`) is an enumeration indicating the type of callback. This can be one of the following:

VarValueChangedCallback — call when value changes.

VarDestroyedCallback — call just before a Var is deleted.

The second argument (`CB`) is a pointer to the callback. The third argument (`clientData`) is a pointer to some additional piece of data to be passed to the callback.

The callback function has two arguments. The first is a pointer to the variable invoking the callback, and the second is the `clientData` pointer. Here is an example of how a callback can be used to automatically update a class attribute when the corresponding variable object is modified.

```
#include <VarClasses.H>
class X {
  double xx;
public:
  X (VarTable *);
  static void xx_changed(Var *, void *);
  // ...
}
// this function is called when the value of variable xx is changed
void X::xx_changed (Var *var, void *p)
{
  ((X*)p)->xx = ((VarNumeric *) var)->get();
}
// constructor for the class X
void X::X (VarTable *tab)
{
  VarNumeric *v_xx = tab->lookupNumeric("xx");
  v_xx->addCallback (VarValueChangedCallback, xx_changed, this);
  // ...
}
```

The callback function `xx_changed()` has to be a static member function in order to correctly pass its address to the variable object. Therefore, the `this` pointer is passed to the callback as client data in order to identify which object of class `X` is to be operated on. The callback uses `get()` to retrieve the new value of the variable and sets the corresponding `double` in the object pointed to by `p`.

7 Using VarLink

Sometimes it is desirable to use the cataloging and I/O features of the Var classes with other data types, for example, the user's favorite matrix class. And sometimes access speed considerations make it more desirable to use a double rather than a VarNumeric within a class. To accommodate these needs, we devised the VarLink and VarData classes.

VarLink is a set of classes that allow the user to interact with an object through member functions or by directly accessing object attributes. Essentially, a VarLink is a variable linked to its value through a pointer to an object and a pointer to a member function or an attribute of that object.

The implementation relies on templates to provide the ability to use nearly any combination of class type and data type. Creating a VarLink requires type information for both the class (`T`) and the value (`D`) being accessed; this information is used to define a two-parameter template class, `VarLink<T,D>`. For example, a VarLink to access a double in the class `myClass` would be created with the following template: `VarLink<myClass, double>`.

Actually, the sole purpose of the VarLink class is to provide an interface to a set of classes that store the values of the pointers mentioned previously. Which of these classes is chosen depends on what arguments are used to identify the object data. The choice depends on whether the value is accessed directly or through member functions and on whether the variable's permission is read only or read/write. This implementation was inspired by the *Envelope/Letter* idiom described by Coplien [4].

Once a VarLink has been created, only the type of value is needed. The VarData class is used to interface with the variable once the variable is defined.

8 Storing the Time History of a Variable

In some simulations, it is necessary to record the variation of a variable over time. We have devised several solutions to this problem.

1. The variation of a variable over time could be recorded by declaring each variable as an array and indexing the arrays during each time step for the simulation. This solution, however, would prohibit arrays from being used for other purposes and/or it would confuse the use of the arrays because they would have too many subscripts.

2. A second method for recording transients is to write variable tables to a file. Although this would be sufficient for storing data, it would not be an adequate solution for programs that need to revisit previous time steps frequently.

3. Yet another method is to store all the transient data in memory. This could be done by making a copy of the variable table for each time step. If all the data for a variable were stored within it, the memory requirement would be $N * S$, where N is the number of time steps to be saved and S is the size of the variable. This, of course, would require more real memory than most systems have for all but the simplest of problems. However, because the strings and equations are not stored in each variable but in separate objects, these value data can be shared between all copies of a variable, greatly reducing the total memory requirement. Only when an attribute of a variable changes is more memory allocated. Reference counting on string and equation objects ensures that new memory is allocated when needed and that old memory is freed when it is no longer used. This method has the advantages that all attributes of a variable may change over time, and the user needs only to index a pointer

to the variable table to change the time step. An easy way to implement this method is to create a VarTable that holds the pointers to the tables for each time step. For example,

```
// read one copy of the variable table from file
VarTable *tab = new VarTable("MyTable");
ifstream strm("MyTable.def");
tab->load(strm);

// create a table to hold transients
VarTable *time = new VarTable("time");
time.add(tab);   // add first time step to time table

// make a copy of the table for each time step and load values
strm >> num_time_steps >> tab;
for (int i=1; i < num_time_steps; i++ ) {
   tab = new VarTable (tab);        // use copy constructor
   time.add(tab);                   // add time step to time table
   strm >> tab;                     // read values from input stream
}
```

Notice that the input stream (strm) needs only to contain the values that have changed from the previous input case. This is similar to FORTRAN namelist input. To loop back through the time steps,

```
for (int i=0; i < time->size(); i++ ) {
   tab = (*time)[i];     // tab points to the current table
   process();            // do some calculation
}
```

If all data in the simulation are transient, then the user should replace tab with MainVarTable in this example. Because the VarTable class dynamically redimensions itself as needed, there is no limit on the number of time steps that can be recorded. In fact, to save memory, the user should make num_time_steps small initially, and add more time steps as needed. The disadvantage of this method is that it requires more memory to be duplicated than is usually necessary.

4. Our final method for recording time histories of data, which will further reduce memory requirements, is to duplicate the array of pointers in each Var class that store the values of the data. For example, the VarNumeric class has four arrays of smart pointers to Eq objects. These arrays are for minimum, maximum, default, and value. In most cases, however, only the value array changes over time. Therefore, there is no need to duplicate the arrays of smart pointers for minimum, maximum, and default values as would be done in the previous method. In this case, each variable contains an additional array of pointers that point to the array of Eqs, strings, or doubles that would be used for each time step. Furthermore, each Var would have to have a method for changing the time step of the variable. A table would have to have a method to change the time step for all the variables in the table. This type of transient support is created for each variable that is declared with the attribute ntime to be greater than one. The Var classes must be compiled with the VAR_TRANSIENT option to use this method.

The best method for storing transient data will depend on the application. If memory is limited, scheme 2 (storage in a file) may have to be used. Scheme 3 requires the most memory. But changing all variables from one time step to the next can be as fast as indexing one pointer. Scheme 4 requires less memory than scheme 3 but takes slightly longer to change time steps, and scheme 4 does not make all the information in a variable time dependent.

9 The Calculator

The calculator is used from within the Eq (equation) class. It converts equations, entered as character strings, into a tree structure that can be evaluated to a double-precision floating point number. This section describes the calculator parser and the tree structure for storing and evaluating expressions.

The calculator supports predefined constants, logarithmic functions, trigonometric functions, scientific notation, if-then-else expressions, and looping. The syntax for these functions is similar to the syntax used in the C programming language. See the reference manual for a complete definition of the calculator's syntax.

Two UNIX utilities, LEX and YACC, were used to implement the calculator program [5]. LEX is a lexical analyzer program generator that can be used for simple lexical analysis of text. The user provides a set of regular expressions and actions to be executed, and LEX generates a C program. YACC is a parsing program generator. The user supplies a context-free grammar, which YACC converts into a set of tables used by an LR(1) parsing algorithm [6]. In addition, the user can specify precedents and associations to remove any ambiguities inherent in the original grammar. YACC generates a C file that can be incorporated into a larger program. LEX and YACC greatly simplify the programming of this parser and make it easy to add new functionality.

An Eq object receives a string as its input. When the Eq object is queried for its value, it passes this string to the lexical analyzer module, which converts it into tokens. The token stream is then passed to the parsing modules, which build up the appropriate parse tree. A pointer to the top node of the parse tree is then returned to the Eq class where it is stored. This top object is called an ntNode. The retrieval function in the Eq class then calls the evaluation function on the ntNode, which returns a double-precision floating point number.

The ntNode class structure is based on a binary tree. Each node represents a number, a binary operator, a unary operator or function call, or a variable in the variable table. The tree gets evaluated by calling the get function on the root node. The get function calls get functions on the left branch followed by the right branch. When those calls return, the operation or function represented by the node is executed. The result of the set of calls is the same as for a Reverse Polish Notation (RPN) stack evaluation.

Additional types of nodes represent conditional statements. The evaluation of a condition determines whether the left or the right branch of the tree is evaluated. An iteration node works similarly.

Finally, there is a node to represent a statement block. This node is actually a collection of several independent parse trees that are evaluated in series. The result of the final parse tree is, by definition, the value of the statement block. In this way, complicated expressions can be simplified by typing a series of subexpressions that assign values to temporary variables. These temporary variables can then be used in the final expression.

Note that equations in variables act more like functions than statements because they are reevaluated each time the value of the variable is used.

10 Graphical User Interface

The variable classes presented here are designed with an interactive user interface in mind. The Var base class stores information such as labels that can be placed in input fields and help information that can be displayed in dialog boxes.

A complete object-oriented graphical user interface framework that was developed on the basis of these variable classes [7]. In this framework, all graphical objects are derived from the base class UIComponent. Some additional functionality was added to the Var base class to store related UIComponents and to notify them when changes are made to a Var. Because multiple UIComponents can be registered with a Var, the same data can be displayed in several places on the screen simultaneously. For example, a VarNumeric can be displayed in a text field and in a plot at the same time. In our framework, both views automatically update when a change is made to the data.

The update process works as follows: The constructor of a view (a UIComponent) is passed a Var (`var`) that holds the data to be displayed. The constructor then calls `var->addView(this)`, which adds the pointer to the UIComponent to a list of views stored in the Var object. If the value of `var` changes, it calls `view->update()` for all the views in its list. The `update()` method retrieves the data from the Var object and changes the display. Note that the list of views in the Var class is allocated only if needed; therefore, there is negligible overhead for the many variables that are never displayed.

The destructor of the view class calls `var->removeView(this)` to remove the view from the Var's view list. The destructor of a Var deletes all the views in its view list. If a user wants to reuse a view for a different Var, the view should be removed from the old Var and added to the new Var. This can save a significant amount of time (the user does not need to destroy and make a new UIComponent), particularly if the view is complex—like a plot or a large text-editing program.

11 Interprocessor Communication

Variables and variable tables may need to be transferred between processes and perhaps between machines in client/server, parallel, and other distributed applications. To facilitate this, the Var classes have built-in member functions for packing and unpacking themselves to and from Parallel Virtual Machine (PVM) message buffers. PVM is a public domain software package for passing messages on a heterogeneous network of Unix computers.

Four functions have been added to all the Var classes. These are `pvm_packall`, `pvm_unpackall`, `pvm_pack`, and `pvm_unpack`. The first two of these functions pack and unpack the complete definition of a Var into and out of a PVM buffer. This includes name, type and size information, help string, label string, and all data. The latter two functions just pack and unpack the Var name and data. If passing a VarTable, these latter two functions will transfer only Var objects that have had their value changed since the last transfer. This can greatly reduce the network's communication costs.

12 Reference Counting

Most simulations will have multiple instances of the same component class, each of which will have a similar copy of a variable table. To help reduce memory requirements, the user creates shallow copies of the variable tables. A *shallow copy* is a copy of an object that duplicates the attributes of the original object but does not duplicate any data "pointed to" by the original object. This allows multiple variables to share the help information, labels, equations (Eqs), and other information

without having to duplicate it. However, it poses a new problem: namely, it is no longer safe to delete any data pointed to by a variable when a variable is destroyed or reassigned.

In addition, it is often convenient to place the same variable into more than one table or to refer to it from more than one component object. When a table or component is destroyed, it is unknown if it is safe to delete the variables contained within the table or component.

Leaving these data undeleted is an acceptable solution for small programs but will most likely pose problems for larger problems. To solve this dilemma, we devised a reference-counting scheme.

This reference-counting scheme uses a *reference count* class and a smart pointer class. The count class contains the number of references and has the methods to increment and decrement that count. Certain operators have been overloaded, and virtual functions are provided to assist with object allocation and to prevent direct access to the machine address of the object. When derived from `RefCount` (either directly or indirectly), a user class inherits the attributes and methods to maintain a reference count. Once reference counting is introduced in a hierarchy, all descendants will have reference-counting characteristics.

12.1 The Pointer Class

The pointer class, `Ptr<T>`, has been given all the semantics of a pointer, and with a few notable exceptions, can be used anywhere a regular pointer is expected. Smart pointers are used to point to dynamically allocated instances of a user class for which reference counting is desired. The process of initialization, assignment, and destruction of smart pointers includes the calling of increment and decrement methods in the appropriate counted object.

Once declared, a `Ptr<T>` can be assigned the address of an instance of class `T` and can be used as if it were a regular pointer to class `T`. For example, if we assume that `userClass` has been derived from the RefCount class, the following code is legal:

```
Ptr<userClass> ptr = new userClass;
ptr->method();          // ptr can be used to access public members
ptr->attribute = 123;
cout << *ptr << "\n";   // ptr may be dereferenced
cout << ptr[0] << "\n"; //    just as a normal pointer

ptr = NULL;             // ptr may be reassigned
```

In this code segment, an instance of `userClass` is created on the heap. Because this is also an initialization of `ptr`, the constructor `Ptr<T>::Ptr(T *)` is called with the address returned by new. This constructor sets the internal pointer to that address and calls a method within the `userClass` instance that increases its reference count from zero to one. After the initialization of `ptr`, accessing members and dereferencing occur as expected.

Before `ptr` is assigned a different value, the reference count of the current `userClass` instance is decremented. In this example, the reference count goes to zero. When this occurs, the object calls the delete operator with its own address as the argument: `delete this;`. The new address is then assigned to `ptr`, and if not `NULL`, the reference count of the new instance is incremented. Note that addresses are checked for equality before performing these actions. However, addresses are not checked for validity; so, be careful!

An object can be deleted only through a call to `decrPtrs()`, which makes the reference count zero. This call can only be done by the reassignment or destruction of a `Ptr<T>`. Once it is determined that the object should be deleted, the state contained in the object is used to decide if `delete this` or `delete[] this` should be issued.

Notice that if the address returned by the new operator is not assigned to any variable, there will be an instance of `userClass` that has nothing referring to it. In this case, there is nothing that can be done to deallocate the memory. This may not seem useful; however, it has the same behavior as a noncounted object whose address is lost. With a counted object, however, a garbage collection scheme could be introduced that looks for allocated objects with no references and deletes them. This would be a distinct advantage over noncounted objects.

When an array of objects is dynamically allocated, the objects are given state information that indicates the location of the object within the array. Regardless of where an object is in the array, calls to `incrPtrs()` and `decrPtrs()` affect a single reference count. In this manner, the array is treated as a single object; thus the entire array can be deleted when there are no longer any references to any of its elements. This is consistent with the C/C++ notion of standard pointers; reference to any one element of the array implies a reference to the entire array.

The position of an element within the array is not important until the result of a `decrPtrs()` call is zero. In order to properly delete the array, the user must send the address of the first element as an argument to `delete[] this`. The method `Ptr<T>::deleteThis()` accomplishes this task. Before calling delete, the state of the object is consulted. For an array, the state indicates the index of the element. This value is used to determine the address of the first element from simple pointer arithmetic. Once the first object is found, `delete[] this` is called.

This smart pointer class is smart only because it sends a message to increment or decrement a counter. All the traditional pointer pitfalls remain. Reference through a `NULL` pointer is chief among them. A list of pitfalls, cautions, and caveats introduced by the smart pointer implementation follows:

- A `Ptr<T>` cannot be used as the argument of `delete`. The argument of `delete` must be a nonconstant pointer; `Ptr<T>` is a class instance. Implicit conversion to `T*` is not allowed. To delete the referenced instance, simply set all `Ptr<T>`'s that reference it to `NULL`.

- To allocate arrays of objects, the user must call an overloaded version of the global new operator. To do this, the placement syntax is employed to pass the size of the class for calculating the number of objects in the array. Notice the `(`*expression*`)` following `new`:

```
Ptr<userClass> p = new( sizeof(userClass) ) userClass[10];
```

- `ptr++` and `ptr--` must be done with care. The increment and decrement operators can take the internal address outside the bounds of an object array. This will not cause immediate problems. If, however, `ptr` goes out of scope or the user attempts to reassign `ptr` while the internal address of `ptr` is outside the array bounds, the results will be undefined. `ptr` will try to decrement the reference count at this point. A segmentation fault is imminent.

- Pointer arithmetic must be done with care. Addition and subtraction will cause immediate effects. The returning of temporary instances of `Ptr<T>` will increment and decrement reference counts. Unlike `ptr++` and `ptr--`, if pointer addition or subtraction results in an address outside the bounds of an array, the effects will be immediate. This must be avoided to prevent a run-time addressing exception.

12.2 Reference Counting Class

In this scheme, a counter is incremented and decremented to keep track of how many times an object is being referenced. As long as the counter is not zero, the object remains in memory; when

the counter goes to zero, the object is deleted. Therefore, ownership is not an issue because an object remains in memory only while it is needed. The smart pointers described previously are used to refer to counted objects.

There are two common ways to implement reference counting. One way is to implement the counter as an independent object. Pointers refer to the counter object, and the counter refers to the actual object. The advantage to this method is the ease with which it is applied to existing classes. This makes it desirable for interfacing with an existing class library. One disadvantage is the additional level of indirection needed to access the object. Another, more important, disadvantage is the inability to take advantage of polymorphism. The counter class is completely unrelated to the class hierarchy of the referenced object; therefore, a pointer to a derived class cannot be used where a pointer to a base class is expected.

The other way is to include the counter directly in the object. Pointers, in this case, refer directly to the object. The advantage here is that polymorphism can be used. Because the actual object is referenced, it is possible to cast the smart pointer to a valid base class. The disadvantage is that run-time type identification (RTTI) must be used for all pointer conversions; this slightly reduces efficiency.[1]

The latter method has been chosen mainly to enable polymorphism. We also determined that the reference counting behavior would be added to a class through inheritance. The class, called RefCount, provides the necessary attributes and methods.

An object can be allocated statically (on the stack) or dynamically (on the heap). In the latter case, `class_name::operator new()` is called to allocate the memory. `RefCount::operator new()` has been overloaded so that it sets a global flag to indicate a heap allocation and then calls the global operator `new()` to allocate the memory. The constructor `RefCount::RefCount()` checks this flag and resets it to its default value. The attribute `RefCount::state` is set accordingly.

When memory is allocated for an array of objects, it must be deallocated as an array. It is, therefore, necessary to differentiate between the two cases. If an array has been allocated, the value of `RefCount::state` will be set to `inst_HEAP_ARRAY +` *element index*. This value is used to determine the address of the first element of an array before calling `::operator delete[](void*)`. For this to occur, the programmer must call a special version of the new operator that expects the `sizeof` the class passed as an additional parameter. At present, the placement syntax for operator `new()` is used to call this special version,[2] as mentioned earlier.

Because there is a fundamental difference between a single object and an array of objects, the counter must be implemented in two ways. For a single object, the count is simply an `int` stored in the object. An array, on the other hand, must also be treated as a single object: references for any element of the array must be tallied in a common count. This is done by making the count within the RefCount class a union of an `int` and a pointer to an instance of the class refs. The latter version of the union is used if an array of objects is allocated. Each element of an array will have the same pointer value in the count union. All increment and decrement operations made on an array element are performed on the common count. The value of `RefCount::state` is used to determine whether the count is an `int` or a pointer.

The current implementation of reference counting requires the overriding of several virtual functions and a special constructor within each class that employs reference counting. Because these are fairly simple, a set of macros is supplied that should be used in defining any class derived from RefCount.

[1] The proposed operator dynamic_cast<*type-name*>(*expression*) will likely reduce this inefficiency.

[2] ANSI/ISO resolutions have introduced the ability to overload operator `new[]()`. When this implementation is available in compilers, using the placement syntax should be unnecessary.

Currently, reference counting is used only for the Eq class. Note that we had added reference counting to all Var classes such that Vars will automatically delete themselves if they are no longer used. However, the Ptr<T> template classes needed for all the different types of Vars caused huge amounts of code to be generated, and this functionality had to be abandoned. We will most likely reimplement reference counting for all Var classes once the size problem has been solved.

13 Concluding Remarks

An object-oriented variable table was created that provides a convenient way to define and catalog data, as well as a user friendly input/output system, a standardized set of access functions, mechanisms for ensuring data integrity, an interpretive language for programming relationships between parameters, and methods for interprocessor data transfer. This object-oriented variable table has proven to be an effective programming methodology for both interactive and batch applications.

Although name lookup functions are provided on the variable tables, they are required to locate a variable the first time only. Subsequently, the variable can be accessed directly. Data integrity is maintained because all variables are objects with their own data-access methods.

All the standard Var classes have built-in support for interprocessor communication and multiview graphical displays. In addition to the standard data types provided, a powerful template class has been added so that other data types can easily take advantage of the variable table class features.

Plans for future work include combining the calculator and input file parsers, resulting in a more powerful programming language that is applicable to more than just the VarNumeric class. Also, as indicated in section 12, automatic reference counting would be added to all Var classes as an optional feature.

References

[1] Kroo, I.: *A New Architecture and Expert System for Aircraft Design Synthesis.* NASA Contract NAG1-1052, 1990.

[2] Curlett, B.P.: *A Generic Graphical User Interface for FORTRAN Programs.* NASA TM-4543, 1994.

[3] Stroustrup, B.: *The C++ Programming Language.* Second Edition, Addison-Wesley, Reading, Massachusetts, 1991.

[4] Coplien, J.O.: *Advanced C++, Programming Styles and Idioms.* Addison-Wesley, Reading, Massachusetts, 1992.

[5] Kernigham, B.W.; and Pike, R.: *The UNIX Programming Environment.* Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[6] Aho, A.V.; Sethi, R.; and Ullman, J.D.: *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[7] Curlett, B.P.; Haas, A.R.; and Naylor, B.A.: *Adaptive Graphical User Interface Framework for Object-Oriented System Simulations.* NASA TM-106790, 1995.

Appendix A—Input Parser Grammar

The grammar for the variable table input files follows:

file-definition:
  $attribute\text{-}list_{opt}$ $variable\text{-}list_{opt}$ $assignment\text{-}list_{opt}$

attribute-list:
  attribute
  attribute-list attribute

attribute:
  `template` = *string* ;
  attribute-name [*number*]$_{opt}$ = $number_{opt}$;
  attribute-name [*number*]$_{opt}$ = $expression\text{-}list_{opt}$;
  attribute-name [*number*]$_{opt}$ = $string\text{-}list_{opt}$;

attribute-name:
  attribute-token
  identifier

attribute-token: one of
  `help label display min max default labels`
  `option true false dir mask value`

variable-list:
  variable
  variable-list variable
  varTable
  variable-list varTable

variable:
  variable-head ;
  variableDECL $identifier\text{-}list;_{opt}$

variableDECL:
  variable-head { $attribute\text{-}list_{opt}$ }

variable-head:
  VarType identifier [*number*]$_{opt}$ [*number*]$_{opt}$

VarType: one of
  `Var VarNumeric VarString VarFile`
  `VarLink VarBoolean VarOption`

varTable:
  varTable-head ;
  `typedef`$_{opt}$ *varTableDECL ;*

varTableDECL:
    varTableDEFN identifile-list$_{opt}$
    varTable-include attribute-list variable-list$_{opt}$

varTable-include:
    varTable-type-name identifier

varTableDEFN:
    varTable-head { *attribute-list$_{opt}$ variable-list$_{opt}$* }

varTable-head:
    `VarTable` *identifier*

assignment-list:
    variable-assignment
    assignment-list variable-assignment

variable-assignment:
    variable-name = *expression-list* ;
    varTable-name { *assignment-list$_{opt}$* }

identifier-list:
    identifier , *identifier*
    identifier-list , *identifier*

expression-list:
    number
    expression
    expression-list , *expression*

string-list:
    string
    string-list , *string*

Appendix B—Calculator Parser Grammar

The grammar for the calculator input files follows:

calculator-expression:
    statement-list
    expr

statement:
    expression-statement
    statement-block

statement-block:
  {*statement-list* }

statement-list:
  $statement\text{-}list_{opt}$ *statement*

expression-statement:
  iteration-expression
  if-then-else
  simple-if
  expr ;

expr:
  variable = *expr*
  variable = `Assign`
  number
  (*expr*)
  function-call
  operator-expression
  logical-expression
  variable

operator-expression:
  expr operator expr
  - *expr*

operator: one of
  * / + - ^

function-call:
  function (*expr*)

function: one of
  `sin cos tan asin acos atan`
  `sqrt exp`

iteration-expression:
  for-loop
  while-loop

for-loop:
  `for` ($expr_{opt}$; *expr* ; $expr_{opt}$) *statement*

while-loop:
  `while` (*expr*) *statement*

logical-expression:
  expr ? *expr* : *expr*

expr logical-operator expr

logical-operator: one of

`== && || < >`

simple-if:

`if (` *expr* `)` *then*$_{opt}$ *statement*

if-then-else:

simple-if `else` *expression-statement*

simple-if `else` *statement-block*

variable:

variable-name `[`*expr*`]`$_{opt}$ `[`*expr*`]`$_{opt}$

variable-name:

identifier qualifier$_{opt}$

qualifier: one of

`.max .min .def`

identifier:

$_{opt}$ `IDENT`

identifier `IDENT`

number:

`NUMBER`

` *expr* `

constant

constant: one of

`pi e true false`

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	November 1994	Technical Memorandum

4. TITLE AND SUBTITLE

Flexible Method for Inter-Object Communication in C++

5. FUNDING NUMBERS

WU–505–69–50

6. AUTHOR(S)

Brian P. Curlett and Jack J. Gould

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135–3191

8. PERFORMING ORGANIZATION REPORT NUMBER

E–9196

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, D.C. 20546–0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

NASA TM–106315

11. SUPPLEMENTARY NOTES

Responsible person, Brian P. Curlett, organization code 2410, (216) 977–7041.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited
Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

A method has been developed for organizing and sharing large amounts of information between objects in C++ code. This method uses a set of object classes to define variables and group them into tables. The variable tables presented here provide a convenient way of defining and cataloging data, as well as a user-friendly input/output system, a standardized set of access functions, mechanisms for ensuring data integrity, methods for interprocessor data transfer, and an interpretive language for programming relationships between parameters. The object-oriented nature of these variable tables enables the use of multiple data types, each with unique attributes and behavior. Because each variable provides its own access methods, redundant table lookup functions can be bypassed, thus decreasing access times while maintaining data integrity. In addition, a method for automatic reference counting was developed to manage memory safely.

14. SUBJECT TERMS

Computer programming; Data structures; Object programs; Software tools

15. NUMBER OF PAGES

26

16. PRICE CODE

A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	

National Aeronautics and
Space Administration
Lewis Research Center
21000 Brookpark Rd.
Cleveland, OH 44135-3191

Official Business
Penalty for Private Use $300

POSTMASTER: If Undeliverable — Do Not Return