Dominic Jesse
9 December 2009

For the final project in Discrete Mathematics, I created an "educational game" based on several concepts from the textbook, using object-oriented programming in C++ to both implement the game and provide a template to create more such games in the future.

Program Concept

The first chapter of Susanna Epp's Discrete Mathematics with Application, 3rd Ed. covers the "logic of compound statements." Section 1.3 covers how to establish arguments as valid/invalid, not by using truth tables (as in earlier sections), but by using basic rules of inference, such as modus ponens, modus tollens, elimination, etc.

This section's exercise set (pp. 41-2) uses several entertaining methods to get students to apply these rules, all forms of logic puzzles. For example, problem 37 presents five statements, then asks the student to find the location of a "pirate treasure." Problem 39 uses a traditional "murder mystery" approach, where six statements, plus logical rules, will yield the identity of the murderer.

General Overview and Program Concept

To design the program, I started with Problem 37 on page 42. The story behind this problem involves the discovery of a cryptic set of clues left behind by a pirate, leading to a hidden treasure. The five clues are as follows:

a. If this house is next to a lake, then the treasure is not in the kitchen.
b. If the tree in the front yard is an elm, then the treasure is in the kitchen.
c. This house is next to a lake.

d. The tree in the front yard is an elm or the treasure is buried under the flagpole.

e. If the tree in the back yard is an oak, then the treasure is in the garage.

This puzzle is a simple one. To solve it on paper, the first step is to break up the compound statements into simple statements, assigning each simple statement to an abstract symbol:

A = This house is next to a lake

B = The treasure is in the kitchen

C = The tree in the front yard is an elm

D = The tree in the back yard is an oak

E = The treasure is buried under the flagpole

F = The treasure is in the garage

With the simple statements defined, one simply rewrites the five clues using these abstract symbols and logical operators:

a. $A \rightarrow {\sim}B$

b. $C \rightarrow B$

c. A

d. $C \vee E$

e. $D \rightarrow F$

Now that the statements are reduced, the problem is solved by establishing the validity of one of the statements B, E, or F, which reveal where the treasure is. The following steps are taken:

1. Statements a and c result in ~B, using modus pollens ($A \rightarrow {\sim}B$, A, ${\sim}B$)

2. The result of Step 1 combines with Statement b, resulting in ~C, using modus tollens (~B, C $\rightarrow$ B, ~C)
3. The result of Step 2 combines with Statement d, resulting in D, using method of elimination (~C, C $\vee$ E, E)
4. The problem is solved when the validity of E is proven (the treasure is buried under the flagpole).

To turn this into an interactive program, one can conceive of a "logical machine," or a program that takes three inputs:

i. The first statement
ii. The second statement
iii. The logical method used to evaluate the two statements and produce a new statement.

It then performs the following operations:

i. Determine that the two statements are compatible
ii. Determine if the logical method is compatible with the two statements

and produces the following output:

i. A new statement based on the two operations

To prevent "clutter," the program should also, once a new statement is created, eliminate the two statements used to create it.

Limits of Grammar and Logic

Ideally, the best way to design an object-oriented program to implement such a game would be to:

a. Create an object representing each "basic statement"

b. Have each statement set as true, false, or unknown
c. After the user enters needed data, have existing programming logical operators determine the output

Unfortunately, while this method is the most elegant, it has several drawbacks that made it impractical.

The first problem is that of grammar. If the output was limited to nothing but abstract symbols, such as A, B, and C, grammar would not be a problem. However, the program needs to output the statements in plain English. Take, for instance, the negated statements that come up, ~B and ~C. As defined:

B = The treasure is in the kitchen

C = The tree in the front yard is an elm

In plain English, the negations translate to:

~B = The treasure is **not** in the kitchen

~C = The tree in the front yard is **not** an elm

Using logical operators, though, they read:

~B = **It is not the case that** the treasure is in the kitchen

~C = **It is not the case that** the tree in the front yard is an elm

In theory, the above two statements are English sentences. However, they are awkward, and while grammatically correct, are not well-written, nor would they be considered understandable.[1] Moreover, This only addresses the problems of *negating* a simple statement; there are still the grammatical rules (including commas, upper- and

[1] It might be possible to program, within each object, the rules for it to negate itself. For example, when negating the B and C, you might program the rule that if negated, you add the word "not" in front of the verb "is." However, not all statements negate that easily. Under these rules, the negation of the statement

lower-casing) involved when one uses conjunction, disjunction, etc. between simple statements.

The second problem is that of logical operation. C++ has logical operators for or, and, and negation (||, &&, !). When dealing with if-then, however, it requires code (if[EXPRESSION]…then…else). It lacks (to my knowledge, at an intermediate programming level) specific rules for other inference rules.[2]

For these reasons, when using C++ OOP, I used an entirely different approach.

Object Design

When creating the objects for the program, I decided to create an object for every statement necessary for the solution of the puzzle, whether or not the statement was simple or compound. Each object would hold a string indicating the clue it held, as well as information on the other objects it can be combined with, and the method used to make this combination work.

Using this method, it is necessary to define every object as soon as the program executes, even ones that the user has not "created" yet. To address this, each object is considered "available" or "not available." An clue-object that is "available" is visible to the user, in that it prints out its clue. Also, an "available" clue can be called by the user to be combined with other objects and methods.

"The man is alive" is "The man is not alive." In plain English, though, the better-written negation would be "The man is dead." While such a system is possible, it is too large for this project.

[2] It is, of course, possible to add functions to C++ that mimic these operations. Earlier in this course, I considered using a library that added PROLOG functions that would allow additional inference rules. However, this effectively would have required learning a new language, which was beyond the scope of this project.

Also, Quine demonstrated that almost all logical statements could be reduced to two symbols – conjunction and negation. For example, according to Quinean logic, the statement $A \rightarrow B$ could be expressed as ~(A ^ ~B). However, learning a whole new method of logic is also beyond the scope of this course.

At the beginning of every “turn,” the program goes through the list of clue-objects. If the clue is “available,” it is printed; if not, it remains invisible. The user is then prompted for three integer inputs: the first two specify the clues the user wants to evaluate, and the third represents the logical method used to combine those clues. If the user gives a correct combination, a new clue becomes visible. If the user enters the correct combination to make the final clue available, the player wins.

Using this framework, one can create an abstract data type (ADT) as follows:

dataTypeName

clueType

domain

Each clueType object contains:

1. A string holding the text of the clue
2. A bool value stating whether the clue is currently accessible or not
3. An integer representing the class instance’s unique identifier
4. An integer representing the unique identifier of the other class instance with which it can be combined
5. An integer specifying the correct logical method necessary to combine the clue with the other
6. An integer representing the unique identifier of the clue that becomes available after a successful combination
7. A bool value stating if the particular object instance wins the game if visible

operations

1. Print the clue (if the object is available)
2. Toggle the object’s availability
3. Check to see whether a particular combination with another object is true (based on domains 2-6 above)
4. Check to see if the object’s status wins the game (meaning it’s available, and if available, is the final clue)
5. Fill all of an object instance’s data members

The ADT can be translated into the following Unified Modeling Language (UML) diagram (the clue is split into two strings to allow better display on the screen):

clueType
-clue1: string -clue2: string -access: bool -ID: int -partner: int -method: int -reveal: int -winner: bool
+printClue(): void +toggleAccess(): void +checkConnection(clueType&, int): int +checkAvailability(): bool +checkWinner(): bool +fillClue(string, string, bool, int, int, int, bool)

With the object defined, the below pseudocode can be used to define the object operations.

```
PRINTCLUE
    IF (access = TRUE) THEN
        PRINT clue
    ENDIF
END

TOGGLEACCESS
    IF (access = TRUE) THEN
        ACCESS := FALSE
    ELSE
        ACCESS := TRUE
    ENDIF
END

CHECKCONNECTION
    PARAMETERS clueType other, integer logicalMethod
    IF (partner = other.ID &&
      ID = other.partner &&
      method = other.method = logicalMethod &&
      access = other.access = TRUE) THEN
        TOGGLEACCESS current clueType object
        TOGGLEACCESS clueType object other
        PRINT affirmative message
        RETURN integer reveal
    ELSE
        PRINT negative message
        RETURN –1
    ENDIF
END
```

NOTE: The checkConnection member function returns an integer equal to the ID of the clue that is revealed on successfully combining the right elements. Back in the main program loop, one uses the toggleAccess member function in conjunction with this integer in order to make the clue available and visible.

```
CHECKAVAILABILITY
    IF (access = TRUE) THEN
        RETURN TRUE
    ELSE
        RETURN FALSE
    ENDIF
END

CHECKWINNER
    IF (winner = TRUE && access = TRUE) THEN
        RETURN TRUE
    ELSE
        RETURN FALSE
END ENDIF
STOP

FILLCLUE
ASSIGN the below eight parameters to an object instance
    string clue1
    string clue2
    bool access
    int ID
    int partner
    int method
    int reveal
    bool winner
END
```

<u>Main Program Design</u>

With the objects defined in the previous section, all that's needed is a main game program loop (a type of game engine), and a function to fill all the objects of the game.

The pseudocode for the primary game loop is as follows:

```
START
DECLARE bool win := FALSE
DOWHILE (win = FALSE)
    PRINT all clueObjects whose availability is TRUE
    PRINT definitions of logical operators with associated integer
    PROMPT for integer clue1
    PROMPT for integer clue2
    PROMPT for integer logicMethod
    CALL CHECKCONNECTION method using objects identified in clue1
    and clue2, using logical method identified in logicMethod
    IF (CHECKCONNECTION returns any integer besides –1) THEN
        CALL TOGGLEACCESS for integer ID returned
    ENDIF
    CALL CHECKWINNER method on all objects
    IF (any object returns TRUE from CHECKWINNER) THEN
        DECLARE bool win := TRUE
        PRINT victory message
    ENDIF
ENDWHILE
STOP
```

The only other major operation is to fill each object with the data appropriate for the “Pirate Treasure” operation. This is simply done using the fillClue function. Appendix A to this report contains the data entered for this game.

<u>Application to other Logical Puzzles</u>

The above program, when compiled as a Win32 console program, in C++, using Microsoft Visual Studio .NET 2003, worked correctly, and the executable program is attached, as well as the C++ code (Appendix B).

If the above framework is correct, then the clueType object and methods should be applicable to many other logic puzzles. The only changes required are:

1. The clueType object instances should be initialized with different data

2. The array of clue objects will be of a different size, as will any number in the main loop that reflects the number of objects
3. If the new problem uses different logical methods, they must be redefined in the main loop

To test this, I used Problem 39 in textbook Section 1.3, on page 42. This logic puzzle is a “murder mystery” problem, with the user manipulating clues to discover who killed “Lord Hazelton.” Perusing the problem easily creates the following basic statements:

A = Lord Hazelton was killed by a blow on the head with a brass candlestick
B = Lady Hazelton was in the dining room at the time of the murder
C = The maid, Sara, was in the dining room at the time of the murder
D = The cook was in the kitchen at the time of the murder
E = The butler killed Lord Hazleton with a fatal dose of strychnine
F = The chauffer killed Lord Hazelton
G = The wine steward killed Lord Hazelton

This in turn can be converted into a table of object data, which is included in Appendix C. The executable file, MurderCase, is attached as well, demonstrating the applicability of the underlying game engine

Future Developments

Although I dismissed the idea earlier, more advanced C++ techniques may offer a simpler solution than of creating an object for every possible outcome prior to program execution. One possibility would be to allow the objects to actually *create* other objects (perhaps objects of a derived class) based on successful input. Using such induction rules would greatly reduce the code needed, and allow much larger puzzles.

By far, though, the greatest improvement would be a major improvement of the programs graphics and user interface. The Win32 console is only good for testing the engine. To make an application end users can enjoy would involve creating a standard user interface screen (instead of scrolling text), as well better input methods. For example, an interface where the user can drag-and-drop clues onto a field for evaluation would be much better than entering integers.

As far as useful applications are concerned, the program, as it is currently written, would make an excellent educational game for students beginning to study logic. If the game did not require the precise name of the logical method for each operation, the program may also be useful as a puzzle game for general users. It can also be embedded in larger games as a "mini-game" to add a puzzle to the gameplay.

Works Cited

Epp, Susanna. Discrete Mathematics with Applications, 3rd Ed. Belmont, Calif. – Brooks/Cole-Thomson Learning, 2004. Section 1-3, pp. 29-43.

Malik, D.S. C++ Programming: From Program Analysis to Program Design, 4th Ed. Boston: Course Technology, 2009.

Quine, Willard Van Orman. Elementary Logic, Revised Ed.. Cambridge, Mass. : Harvard University Press, 1980. Section I-7, “If,” pp. 17-20.

APPENDICES:

Appendix 1 – Object Data Table for Pirate Treasure clues (Excel worksheet)
Appendix 2 – C++ Code for Pirate Treasure program (MS Word document)
Appendix 3 – Object Data Table for Murder Case clues (Excel worksheet)

ALSO ATTACHED are stand-alone, executable files for each game, PirateTreasure.exe and MurderCase.exe.