4.5 Case Study: An 8-Function Calculator

PROBLEM

Write an 8-function calculator program that allows the user to perform addition, subtraction, multiplication, division, exponentiation, base-ten logarithm, factorial, and quit operations.

OBJECT-CENTERED DESIGN

BEHAVIOR. The program will display on the screen a menu of the eight operations, telling the user to enter `+`, `–`, `*`, `/`, `^`, `l`, `!`, or `q` to specify the operation to be performed. The program will read the operation from the keyboard. If the user enters `q`, execution will terminate. Otherwise, the program will display on the screen a prompt for the first operand, which it will then read from the keyboard. If the operation requires two operands, the program will display on the screen a prompt for the second operand, which it will then read from the keyboard. The program will then compute the result of performing the specified operation using the operand(s) provided and output this result. The program should repeat this behavior until the user specifies the `q` operation.

OBJECTS. We can immediately identify the following objects from our behavioral description:

	Software Objects		
Probem Objects	**Type**	**Kind**	**Name**
screen	variable	`ostream`	`cout`
prompt for the first operand	constant	`string`	none
first operand	variable	`double`	*operand1*
keyboard	variable	`istream`	`cin`
menu of operations	constant	`string`	*MENU*
the operation	variable	`char`	*theOperation*
second operand	variable	`double`	*operand2*
the result	variable	`double`	*result*

(We will be studying `string` objects in detail in the Chapter 5. Here we will introduce them by building a `string` object to represent the menu.)

This object list enables us to specify the problem more precisely:

Output: Prompts for input
Input: *operand1*, a real;
theOperation, a character;
and (possibly) *operand2*, a real
Precondition: *theOperation* is one of `+, –, *, /, ^, l, !`, or `q`
Output: the *result* of applying *theOperation* to *operand1* and *operand2*

OPERATIONS. From our behavioral description, we can identify the following operations:

Operation Needed	C++ Operation/Statement
i. Output a character string to the screen (prompts, the menu)	<<
ii. Input a character from the keyboard (the operator)	>>
iii. Input a real value from the keyboard (*operand1*, *operand2*)	>>
iv. Compute the result	none
v. Output a real value to the screen (result)	<<
vi. Repeat the preceding steps, unless the user entered `q` in step ii	forever loop

With the exception of operation iv, each of these operations is provided by a C++ operation or statement as noted at the right of the operations. We will construct a function to perform operation iv. Given such a function, we can organize our operations into the following algorithm:

Algorithm for 8-Function Calculator Problem

1. Display *MENU* via `cout`.
2. Read *theOperator* from `cin`.
3. While *theOperator* is not `'q'`, do the following.
  a. Display a prompt for the first operand via `cout`.
  b. Read *operand1* from `cin`.
  c. If *theOperator* is a binary operator:
    i. Display a prompt for the second operand via `cout`.
    ii. Read *operand2* from `cin`.
  d. Compute *result* using *theOperator*, *operand1*, and *operand2*.
  e. Output *result.*
  f. Read *theOperator* from `cin`.

REFINEMENT. Step 7 of this algorithm involves a nontrivial operation. We will develop a function to perform it using the same design steps we are using to solve the “big” problem.

FUNCTION'S BEHAVIOR. The function should receive *theOperator*, *operand1*, and *operand2* from its caller, and then do the following:

If the Operator is:	The function should:
+	Return the sum of *operand1* and *operand2*
-	Return the difference of *operand1* and *operand2*
*	Return the product of *operand1* and *operand2*
/	Return the quotient of *operand1* and *operand2*
^	Return $operand1^{operand2}$
l	Return the logarithm of *operand1*
!	Compute the factorial of the integer part of *operand1* and return the real number equivalent
invalid	Display an error message and return 0

FUNCTION'S OBJECTS. From this behavioral description, we can identify the following objects:

	Software Objects			
Probem Objects	Type	Kind	Movement	Name
operator	variable	`char`	received	*theOperator*
first operand	variable	`double`	received	*operand1*
second operand	variable	`double`	received	*operand2*
return value	variable	`double`	returned	none

Using this list of object names, we can specify the subproblem this function must solve as follows:

Receive: *theOperator, a character*
operand1, a real value
operand2, a real value
Return: the result of applying *theOperator* to *operand1* and *operand2*

This suggests that we can define the following stub for this function:

```
double OperationResult(char theOperator,
                       double operand1, double operand2)
{
}
```

(Note that we name our operator `theOperator` instead of `operator`, because `operator` is a keyword in C++.)

FUNCTION'S OPERATIONS. From the function's behavioral description, we can identify the following operations:

Operation needed	C++ Operation/statement
i.Sum two reals	`+`
ii.Subtract two reals	`-`
iii.Multiply two reals	`*`
iv.Divide two reals	`/`
v.Raise a real to an integer power	`pow()`
vi.Find the base-ten logarithm of a real	`log10()`
vii.Convert a real to an integer and an integer to a real	type casts
viii.Compute the factorial of an integer	`factorial()`
ix.Display a character string (the error message)	`<<`
x.Select from among i – ix, based on the value of *theOperator*.	`if`

For step x, where we must select one of i – ix, based on *theOperator*, we need selective execution; that is, an `if` statement.

FUNCTION'S ALGORITHM. We can organize these operations into the following algorithm:

Algorithm for Result Calculation Function

1. If *theOperator* is '+':
   Return *operand1* + *operand2*.
2. Otherwise, if *theOperator* is '–'
   Return *operand1* – *operand2*.
3. Otherwise, if *theOperator* is '*'
   Return *operand1* * *operand2*.
4. Otherwise, if *theOperator* is '/'
   Return *operand1* / *operand2*.
5. Otherwise, if *theOperator* is '^'
   Return `pow`(*operand1*, *operand2*).
6. Otherwise, if *theOperator* is 'l'
   Return `log10`(*operand1*).
7. Otherwise, if *theOperator* is '!'
   Return `factorial`(*integer-part-of-operand1*) converted to a real.
8. Otherwise,
   a. Display an error message.
   b. Return 0.

FUNCTION'S CODING. The algorithm for our function can be expressed in C++ as shown in the following program:

Case Study 4.5-1 Result Calculation Function.

```
/* perform() applies operation to operand1 and operand2.
 *
 * Receive:      operation, a character
 *               operand1 and operand2, two doubles
 * Precondition: operation is one of +, -, *, /, ^, l, !
 * Return:       the result of applying operation to operand1
 *                 and operand2
 ******************************************************************/

#include <cmath>           // pow(), log10()
using namespace std;

double perform(char operation, double operand1, double operand2)
{
   if (operation == '+')
      return operand1 + operand2;
   else if (operation == '-')
      return operand1 - operand2;
   else if (operation == '*')
      return operand1 * operand2;
   else if (operation == '/')
      if (operand2 != 0)
         return operand1 / operand2;
      else
      {
        cerr << "Operation Result: division by 0 -- "
                "result undefined!\n";
        return 0.0;
      }
   else if (operation == '^')
      return pow(operand1, operand2);
   else if (operation == 'l')
      return log10(operand1);
   else if (operation == '!')
      return (double)factorial((int)operand1);
   else
   {
      cerr << "OperationResult: invalid operator "
           << operation << " received!\n";
      return 0.0;
   }
}
```

CODING THE PROGRAM. Once the operations are available for each step of the algorithm, we can write a program like the following that encodes the algorithm for the original problem.

Case Study 4.5-2 Program to Solve the Calculator Problem.

```
/* calculator.cpp implements a simple 8-function calculator.
 *
 * Input:  operand1, theOperator, operand2
 * Output: the result of applying theOperator to operand1 and operand2.
 *****************************************************************/

#include <iostream>    // cin, cout, <<, >>
#include <string>      // string
using namespace std;

double perform(char operation, double operand1, double operand2);
int factorial(int n);

int main()
{
   const string MENU = "Enter:\n"
                       "  + for the addition operation\n"
                       "  - for the subtraction operation\n"
                       "  * for the multiplication operation\n"
                       "  / for the division operation\n"
                       "  ^ for the exponentiation operation\n"
                       "  l for the base-10 logarithm operation\n"
                       "  ! for the factorial operation and\n"
                       "  q to quit.\n"
                       "--> ";

   cout << "Welcome to the 8-function calculator!\n\n";

   double operand1,
          operand2,
          result;
   char operator;

   cout << MENU;
   cin >> operator;
   while (operator != 'q')
   {
      cout << "Enter the first operand: ";
      cin >> operand1;

      if (operator == '+' || operator == '-' ||
          operator == '*' || operator == '/' ||
          operator == '^')
```

```
      {
         cout << "Enter the second operand: ";
         cin >> operand2;
      }
      result = perform(operator, operand1, operand2);

      cout << "The result is " << result << endl << endl;

      cout << MENU;
      cin >> operator;
   }
}

/*** Insert the #include directives and the definitions of functions
     factorial() from Figures 4.8 and perform() from
     Figure 4.12 here.  ***/
```

Sample run:

```
Welcome to the 8-function calculator!

Enter:
  + for the addition operation
  - for the subtraction operation
  * for the multiplication operation
  / for the division operation
  ^ for the exponentiation operation
  l for the base-10 logarithm operation
  ! for the factorial operation and
  q to quit
--> /
Enter the first operand: 2
Enter the second operand: 3
The result is 0.666667

Enter:
  + for the addition operation
  - for the subtraction operation
  * for the multiplication operation
  / for the division operation
  ^ for the exponentiation operation
  l for the base-10 logarithm operation
  ! for the factorial operation and
  q to quit
--> ^
Enter the first operand: 2
Enter the second operand: 3
The result is 8

Enter:
  + for the addition operation
  - for the subtraction operation
```

```
  * for the multiplication operation
  / for the division operation
  ^ for the exponentiation operation
  l for the base-10 logarithm operation
  ! for the factorial operation and
  q to quit
--> !
Enter the first operand: 4
The result is 24


Enter:
  + for the addition operation
  - for the subtraction operation
  * for the multiplication operation
  / for the division operation
  ^ for the exponentiation operation
  l for the base-10 logarithm operation
  ! for the factorial operation and
  q to quit
--> q
```

TESTING. The sample runs show only a small group of the tests that must be performed to verify the correctness of the program. In particular, since we are using selective execution, it is possible for one path through the program to contain an error that goes undetected if execution does not follow that path. Thus, each possible execution path through the program (i.e., each operation) must be tested to ensure that it is behaving correctly.