The logic of this sample program is simple:

- Input two integer values from the keyboard
- Perform five basic arithmetic operations
- Display the two data values and the five computed values on the screen.

Considering at the procedural approach first, the logic has three major sections: input the data, do the arithmetic processing, and output the results. Thinking about program logic as a series of sections or a list of instructions is fundamental to the procedural approach. Programmers using the procedural approach see the logic of a program as an algorithm, that is, a set of well-defined steps that the computer will perform. The focus of the design is the procedures needed to process the data. The procedures are implemented as functions and the data items are passed to functions as arguments.

In object-oriented programming (OOP), the design focuses on the objects which encapsulate the data and functions needed to process the data. The first step in the OOP approach is deciding on the class or classes of objects to use. The technique that is used here is to write a narrative description of the problem. The nouns used in the narrative give hints about the classes and the data members. The verbs give hints about the member functions. Keep in mind that there will not be a one-to-one correspondence between each noun or verb and some class or class member, but this technique will serve as a starting point.
The following is another description of our simple program:

The data consists of two integer numbers. The program must input the two numbers from the keyboard. The program must add the two numbers, subtract the two numbers, multiply the two numbers, divide the two numbers, and raise the first number to the second number. Finally, the program must output all values.

Fully admitting that the above narrative is as contrived as the sample program, the noun/verb search is an acceptable starting point. Thus, the design of the OOP approach to our simple program uses a single class Numbers, having two data members and seven member functions:

**Numbers**

<table>
<thead>
<tr>
<th>Data</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>first integer number</td>
<td>get the numbers</td>
</tr>
<tr>
<td>second integer number</td>
<td>show the numbers</td>
</tr>
</tbody>
</table>

get the numbers
show the numbers
add the numbers
subtract the numbers
multiply the numbers
divide the numbers
get the power of the numbers
In the next step, more details are given to the above class description.

**Numbers**

**Data**
- first integer number: First value to be entered at keyboard.
- second integer number: Second value to be entered at keyboard.

**Functions**
- get the numbers: Input the two numbers from the keyboard.
- show the numbers: Display the two numbers on the screen.
- add the numbers: Compute and return the sum of the two numbers.
- subtract the numbers: Compute and return the difference.
- multiply the numbers: Compute and return the product.
- divide the numbers: Compute and return the quotient.
- get the power of the numbers: Do exponentiation and return result.

Accessibility of the data members and member functions must now be determined. At this simple level, there are two levels of accessibility: **public** and **private**. Public access means that the members, both data and functions, can be accessed from any location in the program where an object is declared. Private access means that the members, both data and functions, can be accessed only by class member functions. In general, make member functions public so that they can be invoked anywhere where an object has been declared, and make data members private so that the only legal access is through member functions. Usually, public is listed first in the class declaration because the user of the class is most interested in the public interface. Recall that the user of the class is usually **main()**.

The coding of the program is next. This process can be divided into three steps. First, code the class declaration. Recall that the class declaration is the prototype for the objects of the class, including the prototypes for the member functions and declarations of the data members. The second step is code the definition of the member functions. Of course, this is the actual code of the member functions. In step one, the programmer concentrates on the design of the class without having to think about the details of how the member functions are to be coded in step two. The third and final step is code the program, usually **main()**, that will use the class. Notice the lack of details in **main()**.
Visualize the Numbers class and the object data as follows:

```plaintext
class Numbers
    Member Functions (Methods)
        get_numbers()
        show_numbers()
        add()
        subtract()
        multiply()
        divide()
        get_power()

Objects: (Instances of Numbers)

data

    first_num
    15

    second_num
    5
```

Recall that there is only one copy of the member functions to be shared by all objects of the class. Note that there is only one object, `data`, of type `Numbers`. Thus all messages are sent to `data`. For example, the `add` message is sent to `data` in line 54. Remember that the `add()` member function returns the sum of the two data members.

Note that in the procedural approach, the data is basically independent of the code. Each program that is to process this data must define how the data is to be used, resulting in redundant code. With the OOP approach, the data and the program instructions, that is, the member functions, are combined into objects.
// Procedural: Simple program using the procedural approach

// This program inputs two integer values from the keyboard. It then
// performs five arithmetic operations on the two data values, and
// outputs all values to the screen.

#include <iostream>  // for cin, cout
#include <cmath>     // for pow

using namespace std;

int main()
{
    int first_num, second_num;

    // Enter two numbers from keyboard (Input Section)
    cout << "Enter the first number:\n";
    cin >> first_num;
    cout << "Enter the second number:\n";
    cin >> second_num;

    // Perform arithmetic processing (Processing Section)
    int sum, difference, product, quotient, power;
    sum = first_num + second_num;
    difference = first_num - second_num;
    product = first_num * second_num;
    quotient = first_num / second_num;
    power = pow(first_num, second_num);
        // double pow(double base, double exponent);

    // Print results (Output Section)
    cout << "\nData values are:\n";
    cout << first_num << " and " << second_num << endl << endl;
    cout << "Computed values are:" << endl;
    cout << "Sum .......... " << sum << endl;
    cout << "Difference ... " << difference << endl;
    cout << "Product ...... " << product << endl;
    cout << "Quotient ..... " << quotient << endl;
    cout << "Power ........ " << power << endl;
    cout << "\nEnd of Arithmetic Program\n";
    return 0;
}
Output:
Enter the first number:
15
Enter the second number:
5
Data values are:
15 and 5
Computed values are:
Sum ............ 20
Difference ... 10
Product ...... 75
Quotient ..... 3
Power .......... 759375
End of Arithmetic Program
// OOPApproach: Simple program using OOP approach

#include <iostream> // for cin, cout
#include <cmath> // for pow

using namespace std;

// Declare Numbers class

class Numbers
{
public:
    void get_numbers(); // Prototypes for member functions
    void show_numbers();
    int add() { return first_num + second_num; } // inline methods
    int subtract() { return first_num - second_num; }
    int multiply() { return first_num * second_num; }
    int divide() { return first_num / second_num; }
    int get_power() { return pow(first_num, second_num); }

private:
    int first_num; // Declarations for data members
    int second_num;
};

// Define member functions for Numbers class

void Numbers::get_numbers() // Enter two numbers from keyboard
{
    cout << "Enter the first number:\n";
    cin >> first_num;
    cout << "Enter the second number:\n";
    cin >> second_num;
}

void Numbers::show_numbers() // Display the two data values on the screen
{
    cout << "\nData values are:\n";
    cout << first_num << " and " << second_num << endl << endl;
}
```cpp
int main()
{
    Numbers data; /* Declare data to be object of first_num and second_num of the data object. */
    data.get_numbers(); // Send the get_numbers message to data
    data.show_numbers(); // Send the show_numbers message to data
    cout << "Computed values are:" << endl;
    cout << "Sum ........... " << data.add() << endl;
    cout << "Difference ... " << data.subtract() << endl;
    cout << "Product ...... " << data.multiply() << endl;
    cout << "Quotient ..... " << data.divide() << endl;
    cout << "Power ........ " << data.get_power() << endl;
    cout << "\nEnd of Arithmetic Program\n";
    return 0;
}
```

**Output:**

Enter the first number:
15
Enter the second number:
5

Data values are:
15 and 5

Computed values are:
Sum ........... 20
Difference ... 10
Product ...... 75
Quotient ..... 3
Power ........ 759375
End of Arithmetic Program

**Stack:**

```
data | first_num
|------|------
|      | 15   |
second_num
|------|------
|      | 5    |
main()```


The private data members of an object can only be accessed by the member functions (methods) that are part of the class. Thus, the data is “hidden” from parts of the program that are not users of the class. Recall that member functions can only be accessed through an object, that is, by sending a message to an object. Thus, if a function has not declared an object variable, there is no object to receive a message to invoke a method which can access the private data of the object. This bundling of the data members and member functions is also called encapsulation because the data and the functions that manipulate the data are put together in a capsule or a class declaration.

In C++, the above ideas are directly related to class member access control, that is, public or private. A new version of the declaration for the Numbers class, making all members of the class public, is shown below:

```cpp
class Numbers
{
    public:
        void get_numbers(); // Prototype for member functions
        void show_numbers();
        int add();
        int subtract();
        int multiply();
        int divide();
        int get_power();
        int first_num, second_num; // Data members
};
```

A class member that is declared as public can be accessed by any user of the class, that is, any function that declares one or more objects of the class. Recall that the user of a class is typically main(). Public members are accessed using the dot member operator. To illustrate some typical references, a partial main is shown below:
int main()
{
    Numbers data, pair; // Declare data and pair to be objects of class Numbers
    data.get_numbers(); // Send get_numbers message to data
    pair.get_numbers(); // Send get_numbers message to pair
    cout << data.first_num; // Output 1st number of data
    cout << data.second_num; // Output 2nd number of data
    cout << pair.first_num; // Output 1st number of pair
    cout << pair.second_num; // Output 2nd number of pair
    ...
}

Assume that all members of class Numbers are now public. Note that any public member, either a data member and a member function, can be accessed using the following syntax:

    object’s name.member name

Thus, main() or any function that is not part of a class declaration (a free function) can declare an object variable and have direct access to the class public members, and no data is hidden.

The original declaration for Numbers is shown below:

    class Numbers
    {
        public:
            void get_numbers(); // Prototype for member functions
            void show_numbers();
            int add();
            int subtract();
            int multiply();
            int divide();
            int get_power();
        private:
            int first_num, second_num; // Data members
    }

Now the data members are private. A class member that is declared as private can only be accessed by members of the class, that is, by the member functions. Thus in the partial main shown above, the four cout’s would fail because they reference private data members of objects, data and pair, and main() is not a member function. If you look at the complete program listing, you will see that there is a member function, show_numbers(), that is responsible for printing
the two data members, first_num and second_num. Recalling that only member functions can access private members, the four cout’s must be replaced by:

```cpp
    data.show_numbers();  // Send the show_numbers message to data
    pair.show_numbers();   // Send the show_numbers message to pair
```

The latter declaration for Numbers is typical, in that the member functions are declared to be public and the data members are declared to be private. Only member functions can directly access the private data members. Thus, the data is hidden from any part of the program that does not use the Numbers class, that is, where an object of type Numbers has not been declared. Recall that the public member functions can only be accessed by users of the class. With this typical declaration of members, at least one member function must be an accessor function, that is, a function that allows the user to read, but not modify, the private data members of the class. Numbers::show_numbers() is an accessor function. We have the concept of controlled access where the private data members are accessed by one or more member functions, and the rest of the program must go through these member functions to modify or even look at the data. This yields complete control over what parts of the program can access and modify the private data members. A mutator function is a member function that modifies the private data members.

Data hiding is also known as information hiding. It might even be thought of as complexity hiding. Encapsulation conceals the internal workings of the function members as well as concealing the data members. The user of a class does not have to know the details of the implementation, but the nature of the object and its interface. For example, the user of the Numbers::get_power() member function does not need to be aware that get_power() uses the function pow() or how pow() works, but only needs to understand how to use get_power(). For a more general example, cin and cout are actually objects that are declared in the iostream file. To use these objects, a programmer must not understand the internal workings of the class members, but merely the operations performed by the class and how to interface with the class.

At this point, you might be thinking that invoking a member function is similar to invoking a procedure in a procedural language. By only knowing how to interface with the procedure, a C
programmer could use a sort function without understanding the details of the sort algorithm. Message passing is very similar to a function call. In both cases, there is a request for action and a set of well-defined steps for carrying out the action. However, there are two differences. An object receives the message. In a procedure call, there is no receiver. The second difference is the interpretation of the message is dependent upon the receiver. For example, we might have another class `ComplexNumber`.

```c
Numbers data; // Declare data to be a Numbers object
ComplexNumber comp; // Declare comp to be a ComplexNumber object
data.add(); // Send the add message to object data
comp.add(); // Send the add message to object comp
```

In the above code, note that the `add` message is being sent to both `data`, an object of type `Numbers`, and to `comp`, an object of type `ComplexNumber`. Just as class `Numbers` has an `add()` function to add two integers, class `ComplexNumber` would have a member function `add()` that would add two complex numbers. Since `data` is type `Numbers`, `Numbers::add()` would be invoked first. Since `comp` is type `ComplexNumber`, `ComplexNumber::add()` would be invoked second. Thus, the message `add` has two interpretations depending on the class of the object. This is a form of `polymorphism`.

Another advantage of information hiding is the ability to change the private members of a class declaration and definition without any changes to the code that uses the class. For example, we could redesign the `Numbers` class to consist of two `float` numbers rather than two `int` numbers. The code in main, the user of the class, would not have to be changed at all, but `Numbers::add()` would be modified to add two float numbers rather than two int numbers and return a float rather than an int.

Class objects are considered to be self-contained. We should think that the objects take care of handling the data structures and the complexities of manipulating the class objects. The self-contained nature of encapsulated objects encourages the reuse of already developed code. Encapsulation also reduces the interconnectedness among software systems which is one of the principal causes of software complexity in the procedural approach.