CS 210 – Fundamentals of Programming I
Analysis and Design Style Guideline

The purpose of the analysis and design is to outline a program before it is implemented. There are several reasons why this required. First, spending time writing out the analysis and design can greatly decrease the time it takes to implement a solution to a problem. The programmer is able to concentrate on what needs to be done to solve the problem rather than on the intricacies of the compiler or the syntax of the language of implementation. Second, often the person who designs the program is not the person who implements the program. As such, the analysis and design clearly documents what is needed to solve the problem without using code-specific jargon that the programmer may not be familiar with. In some cases, the implementation language is chosen after the analysis and design is completed.

Analysis and Design for Main Programs and Functions

An analysis describes the data objects needed to solve the problem. It includes a description of the object, its type, whether it is a constant or a variable, and its name in the program. The analysis corresponds to the variable and constant declarations of a program. The analysis also describes the interfaces (the types of the argument and returned objects) of the functions used in the program (called function prototypes in C++), as well as the variables and constants that must be declared within the function.

A design is the "outline" of the computational steps. It is often used synonymously with the term algorithm. There is a design for each function in a program as well as the main program itself. When implementing a function, the design corresponds to the executable statements needed to compute the result. The analysis and design format in this document is a shortened and modified version of the Object-Centered Design approach used in Adams, Leestma, and Nyhoff, C++: An Introduction to Computing, 2/e.

Analysis for Main Programs and Functions

The main purpose of the analysis is to specify the objects that the program uses. Objects are data items that have values. They usually correspond to the nouns in the problem statement. For each object, we need to specify a description (what it is), its type (the class of values it might be), whether it is a constant or variable (its kind), and perhaps a name. In addition, for the objects used in functions, we need to specify the object's movement as received (in), returned (out), passed back (out), or local (no movement).

There should an analysis for the main program, and each subprogram (function). It should be in the form of a table. For example, a main program analysis might look like:

<table>
<thead>
<tr>
<th>Objects</th>
<th>Type</th>
<th>Kind</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fahrenheit temperature</td>
<td>double</td>
<td>variable</td>
<td>fahrenheit</td>
</tr>
<tr>
<td>Celsius temperature</td>
<td>double</td>
<td>variable</td>
<td>celsius</td>
</tr>
</tbody>
</table>

A function analysis might look like:

<table>
<thead>
<tr>
<th>Objects</th>
<th>Type</th>
<th>Kind</th>
<th>Movement</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>a verse of &quot;The Farmer in the Dell&quot;</td>
<td>string</td>
<td>constant</td>
<td>local</td>
<td>verse</td>
</tr>
<tr>
<td>a phrase</td>
<td>string</td>
<td>variable</td>
<td>received</td>
<td>restOfLine</td>
</tr>
</tbody>
</table>

The only difference between a function analysis and the main program analysis is that the main program does not have a movement column, since it does not take any arguments and its return value is not significant to the problem statement.

All named objects should be listed, except the screen and the keyboard (cout and cin, respectively, in C++), since they appear in nearly every program. Unnamed objects (such as prompt and output strings) do
not need to be listed unless used as a return value of a function. Global constants should be listed in the main program analysis.

**Design for Main Programs and Functions**

A design should list the main steps needed to solve the problem at hand. The steps should be numbered and the substeps should be subnumbered. Each step should be explained in the language of the problem. For the main function, this means that the steps are in the language of the original problem statement. For other functions, the steps are in the language of the subproblem being solved which is at a more detailed level than at the main program level. That is, we start describing how to compute the data item in question. But at the same time, if at this level, a step is better described abstractly rather than concretely, one would use a function to compute it and say so accordingly.

The format for various design constructs are as follows:

- **Input from keyboard**
  1. Read in list of items to input

- **Input from a file stream named inStream**
  1. Read in list of items to input from inStream

- **Output to screen**
  1. Display/print list of items to output

- **Output to a file stream named outStream**
  1. Display/print list of items to output to outStream.

- **Initialization**
  1. Initialize aVariable to initial value.

- **Computation using expression**
  1. Compute aVariable = expression

  Note that an expression can be a single literal or variable. The computation should be expressed in English or standard mathematical notation, not in programming language notation. In particular, the use of C++ operators like ++ or *= is not appropriate. Instead, these computations might be expressed as

  1. Increment aVariable
  2. Multiply aVariable by expression

- **Computation using a function that returns a value**
  1. Compute aVariable = ComputeSomething (list of arguments)

- **Computation using a function that does not return a value**
  1. Compute something using ComputeSomething (list of arguments)
● Function return

1. Return expression

Again note that the return expression can be a single literal or variable.

● Branching selection. The condition and else are numbered consecutively. The bodies subnumbered accordingly.

1. If condition then
   1.1. Do something
2. Else
   2.1. Do something else

Note that the condition can be any expression resulting in a boolean value. Also note that the else portion is optional.

● Multi-way branch selection - each branch is numbered consecutively with bodies subnumbered accordingly.

1. If condition1 then
   1.1. Do something1
2. Else if condition2 then
   2.1. Do something2
3. Else
   3.1. Do something else

● Case selection - each case is subnumbered consecutively with case actions further subnumbered.

1. Select on anExpression
   1.1. Case case 1 value
      1.1.1. Do case 1
   1.2. Case case 2 value
      1.2.1. Do case 2

● Counted repetition - header indicates loop control variable, starting value, stopping value, and step size (assumed to be 1 if omitted). Body is subnumbered.

1. For lcv from initial value to final value by step size
   1.1. Do something

Note that the construct implies that the loop control variable is automatically initialized and updated for each iteration.

● Pre-test conditional repetition (including loop control variable initialization and update). Body is subnumbered.

1. Initialize lcv
2. While test lcv do
   2.1. Do something
   2.2. Update lcv
Post-test conditional repetition. The do and while are numbered consecutively. Body is subnumbered.

1. Do
   1.1. Do something
2. while test lcv

Indefinite repetition with termination condition. Body is subnumbered.

1. Loop
   1.1. Do something1
   1.2. If termination condition then
       1.2.1. Terminate loop
   1.3. Do something2

Example Analysis and Design for Simple Program

Problem Statement
The New Telephone Company has the following rate structure for long-distance calls:

- The regular rate for a call is $0.40 per minute.
- Any call started at or after 6:00pm (1800 hours) but before 8:00am (0800 hours) is discounted 50 percent.
- Any call longer than 60 minutes receives a 15 percent discount on its cost (after any other discount is subtracted).
- All calls are subject to a 4 percent federal tax on their final cost.

Write a program that reads the start time for a call based on a 24-hour clock and the length of the call. The gross cost (before any discounts or tax) should be printed, followed by the net cost (after discounts are deducted and tax is added).

For the purposes of this example, we will break this program up into two pieces, the main program and a function that computes the net cost of the phone call.

Main Program Analysis

<table>
<thead>
<tr>
<th>Objects</th>
<th>Type</th>
<th>Kind</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>regular rate (.40)</td>
<td>double</td>
<td>constant</td>
<td>REGULAR_RATE</td>
</tr>
<tr>
<td>time of day the call started in military time</td>
<td>int</td>
<td>variable</td>
<td>startTime</td>
</tr>
<tr>
<td>number of minutes the call lasted</td>
<td>int</td>
<td>variable</td>
<td>callLength</td>
</tr>
<tr>
<td>cost of call before discounts and tax</td>
<td>double</td>
<td>variable</td>
<td>grossCost</td>
</tr>
<tr>
<td>cost of call after discounts and tax</td>
<td>double</td>
<td>variable</td>
<td>netCost</td>
</tr>
</tbody>
</table>

Main Program Design

1. Read in startTime and callLength
2. Compute grossCost = REGULAR_RATE • callLength
3. Compute netCost = ComputeNetCost (startTime, callLength, grossCost)
4. Display gross cost and net cost

Function ComputeNetCost Analysis
We specified in the main program design that we would use a function ComputeNetCost. Here is a possible analysis of this function:
<table>
<thead>
<tr>
<th>Objects</th>
<th>Type</th>
<th>Kind</th>
<th>Movement</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>night discount (.50)</td>
<td>double</td>
<td>constant</td>
<td>local</td>
<td>NIGHT_DISCOUNT</td>
</tr>
<tr>
<td>length discount (.15)</td>
<td>double</td>
<td>constant</td>
<td>local</td>
<td>LENGTH_DISCOUNT</td>
</tr>
<tr>
<td>federal tax rate (.04)</td>
<td>double</td>
<td>constant</td>
<td>local</td>
<td>FED_TAX_RATE</td>
</tr>
<tr>
<td>start of night time (1800)</td>
<td>int</td>
<td>constant</td>
<td>local</td>
<td>NIGHT_START</td>
</tr>
<tr>
<td>end of night time (800)</td>
<td>int</td>
<td>constant</td>
<td>local</td>
<td>NIGHT_END</td>
</tr>
<tr>
<td>long call length (60)</td>
<td>int</td>
<td>constant</td>
<td>local</td>
<td>LONG_CALL</td>
</tr>
<tr>
<td>time call started</td>
<td>int</td>
<td>variable</td>
<td>received</td>
<td>starts</td>
</tr>
<tr>
<td>number of minutes the call lasted</td>
<td>int</td>
<td>variable</td>
<td>received</td>
<td>length</td>
</tr>
<tr>
<td>gross cost of the call</td>
<td>double</td>
<td>variable</td>
<td>received</td>
<td>gross</td>
</tr>
<tr>
<td>net cost of the call</td>
<td>double</td>
<td>variable</td>
<td>returned</td>
<td>net</td>
</tr>
<tr>
<td>amount of tax</td>
<td>double</td>
<td>variable</td>
<td>local</td>
<td>tax</td>
</tr>
</tbody>
</table>

**Function ComputeNetCost Design**

1. If start < NIGHT_END or start ≥ NIGHT_START then
   1.1. Compute net = gross * NIGHT_DISCOUNT
2. Else
   2.1. Compute net = gross
3. If length > LONG_CALL then
   3.1. Compute net = net - (net • LENGTH_DISCOUNT)
4. Compute tax = net • TAX_RATE
5. Compute net = net + tax
6. Return net

**Analysis and Design of Classes**

The analysis and design of a class consists of a specification of its attributes and its operations. The specification of the attributes of a class is a table of the attribute objects not unlike the analysis portion of a main program or function. It includes the description, type, and name of the attribute object, but does not need to include the kind or movement of the object. The attributes are assumed to part of the analysis of any member operation.

The specification of the operations of a class is a list of function analyses and designs for each member or friend function of the class (including overloaded operators). The operations should be listed in groups according to what the operations do. For example, constructors, and destructors, accessors and other read-only operations, mutators, I/O functions, and overloaded operators. Friend functions should be listed after the member functions. Finally, any free overloaded functions should be listed last. Since the attribute objects are accessible by all member and friend functions, attributes are not listed again in the analyses of these functions, unless being returned, but are to be used directly in the designs. For example, the analysis and design for the Temperature class from the textbook would be written as follows:
Specification for Temperature class
Attributes - data stored in the object

<table>
<thead>
<tr>
<th>Objects</th>
<th>Type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of degrees</td>
<td>double</td>
<td>myDegrees</td>
</tr>
<tr>
<td>Temperature scale</td>
<td>char</td>
<td>myScale</td>
</tr>
</tbody>
</table>

Operations - analyses and designs of the class functions
(This is not a complete set of operations, but it gives an idea of format)

- Default constructor
  Analysis - no objects
  Design - what are the steps to solve this problem?
  1. Initialize myDegrees to 0.0
  2. Initialize myScale to 'C'

- Explicit-value constructor
  Analysis - what data is received, passed back, returned, or local?

<table>
<thead>
<tr>
<th>Objects</th>
<th>Type</th>
<th>Kind</th>
<th>Movement</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial degrees</td>
<td>double</td>
<td>variable</td>
<td>received</td>
<td>initialDegrees</td>
</tr>
<tr>
<td>Initial scale</td>
<td>char</td>
<td>variable</td>
<td>received</td>
<td>initialScale</td>
</tr>
</tbody>
</table>

Design - what are the steps to solve this problem?

1. If islower (initialScale) then
   1.1 Convert to uppercase, initialScale = toupper (initialScale)
2. Initialize myDegrees to initialDegrees
3. Select on initialScale
   3.1 Case of 'F', 'C', 'K'
      3.1.1 Initialize myScale to initialScale
   3.2 Default case
      3.2.1 Print error message
      3.2.2 Initialize myScale to 'C'

- Degrees- returns the number of degrees of this temperature
  Analysis - what data is received, passed back, returned, or local?

<table>
<thead>
<tr>
<th>Objects</th>
<th>Type</th>
<th>Kind</th>
<th>Movement</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of degrees</td>
<td>double</td>
<td>variable</td>
<td>returned</td>
<td>myDegrees</td>
</tr>
</tbody>
</table>

Design - what are the steps to solve this problem?

1. Return myDegrees

- Scale- returns the scale of this temperature
  Analysis - what data is received, passed back, returned, or local?
<table>
<thead>
<tr>
<th>Objects</th>
<th>Type</th>
<th>Kind</th>
<th>Movement</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scale of temperature</td>
<td>char</td>
<td>variable</td>
<td>returned</td>
<td>myScale</td>
</tr>
</tbody>
</table>

Design - what are the steps to solve this problem?

1. Return myScale

- Fahrenheit - return the Fahrenheit equivalent of this temperature

Analysis - what data is received, passed back, returned, or local?

<table>
<thead>
<tr>
<th>Objects</th>
<th>Type</th>
<th>Kind</th>
<th>Movement</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fahrenheit equivalent</td>
<td>Temperature</td>
<td>variable</td>
<td>returned</td>
<td>-----</td>
</tr>
</tbody>
</table>

Design - what are the steps to solve this problem?

1. Select on myScale
   1.1 Case 'F' (already in Fahrenheit)
      1.1.1 Return Temperature (myDegrees, 'F')
   1.2 Case 'C'
      1.2.1 Return Temperature (myDegrees * 1.8 + 32.0, 'F')
   1.3 Case 'K'
      1.3.1 Return Temperature ((myDegrees - 273.15) * 1.8 + 32.0, 'F')

- friend operator== - return true if two temperatures are equal, false otherwise

Analysis - what data is received, passed back, returned, or local?

<table>
<thead>
<tr>
<th>Objects</th>
<th>Type</th>
<th>Kind</th>
<th>Movement</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left operand</td>
<td>Temperature</td>
<td>variable</td>
<td>received</td>
<td>leftOperand</td>
</tr>
<tr>
<td>Right operand</td>
<td>Temperature</td>
<td>variable</td>
<td>received</td>
<td>rightOperand</td>
</tr>
<tr>
<td>Result of compare</td>
<td>bool</td>
<td>variable</td>
<td>returned</td>
<td>-----</td>
</tr>
<tr>
<td>Same scale temperature</td>
<td>Temperature</td>
<td>variable</td>
<td>local</td>
<td>localTemp</td>
</tr>
</tbody>
</table>

Design - what are the steps to solve this problem?

1. Select on leftOperand.myScale (find rightOperand equivalent in same scale)
   1.1 Case 'C'
      1.1.1 localTemp = rightOperand.Celsius()
   1.2 Case 'F'
      1.2.1 localTemp = rightOperand.Fahrenheit()
   1.3 Case 'K'
      1.3.1 localTemp = rightOperand.Kelvin()
2. Return leftOperand.myDegrees == localTemp.myDegrees

- friend operator+ - return new temperature with added degrees

Analysis - what data is received, passed back, returned, or local?
Design - what are the steps to solve this problem?

1. Return Temperature (leftOperand.myDegrees + rightOperand, leftOperand.myScale)

- friend operator<< - Output degrees and scale to an output stream

<table>
<thead>
<tr>
<th>Objects</th>
<th>Type</th>
<th>Kind</th>
<th>Movement</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output stream to write to</td>
<td>ostream</td>
<td>variable</td>
<td>received, passed back, &amp; returned</td>
<td>outStream</td>
</tr>
<tr>
<td>Temperature to output</td>
<td>Temperature</td>
<td>variable</td>
<td>received</td>
<td>theTemp</td>
</tr>
</tbody>
</table>

Analysis - what data is received, passed back, returned, or local?

- friend operator>> - read a temperature from an input stream

<table>
<thead>
<tr>
<th>Objects</th>
<th>Type</th>
<th>Kind</th>
<th>Movement</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input stream to read from</td>
<td>istream</td>
<td>variable</td>
<td>received, passed back, &amp; returned</td>
<td>inStream</td>
</tr>
<tr>
<td>Temperature to read into</td>
<td>Temperature</td>
<td>variable</td>
<td>passed back</td>
<td>theTemp</td>
</tr>
<tr>
<td>Input degrees</td>
<td>double</td>
<td>variable</td>
<td>local</td>
<td>inDegrees</td>
</tr>
<tr>
<td>Input scale</td>
<td>char</td>
<td>variable</td>
<td>local</td>
<td>inScale</td>
</tr>
</tbody>
</table>

Design - what are the steps to solve this problem?

1. Write myDegrees and myScale to outStream
2. Returne outStream

- friend operator>> - read a temperature from an input stream

1. Read in inDegrees and inScale from inStream
2. If islower (inScale) then
   2.1 Convert to uppercase, inScale = toupper (inScale)
3. Select on initialScale
   3.1 Case of 'F', 'C', 'K'
      3.1.1 Set theTemp.myDegrees to inDegrees
      3.1.2 Set theTemp.myScale to inScale
   3.2 Default case
      3.2.1 Set input stream fail bit using inStream.set_state(ios::fail)