An Introduction to Computer
Problem Solving
and the
Programming Process

CS1400 Course Reader
Fall 2006

by Linda DuHadway and Mary Veronica Kolesar
I. PROGRAMMING AND PROBLEM SOLVING

Computer science is about problem solving, be it writing a program (software) to solve a problem or designing a new method of solving a problem or designing physical devices (hardware) that can follow the instructions in a program to solve a problem.

Problem solving is not new to you. You have been solving problems since you were an infant. Think of some of the problems you solved in the last week. Many of them you solved without much conscious thought: getting ready for the day or driving from home to the video store. Other problems required more thought and planning, such as moving your belongings from your summer residence to the residence you will occupy while you go to school or how to meet new friends. And there are always those dreaded “word problems”: you want to go to McBurp’s for a BurpBurger, fries and a drink that cost $3.97 and to a movie that costs $6.00. You have $10.00. Exactly how much will you have to borrow from your little brother so you’ll have enough to buy a small popcorn ($1.25) when you get there?

Whoever you are or whatever your interests, solving problems is an ordinary part of your world. And, since programming is really nothing more than solving a problem yourself and then telling a computer what to do to produce the solution, there is much you already know about the process.

People approach problem solving in different ways. By thinking about the characteristics of “successful” and “less successful” problem solvers, we can gain an insight into how we can become more successful problem solvers.

Successful problem solvers:

- have or acquire the necessary subject matter knowledge
- have a positive, objective attitude, viewing the activity as a learning opportunity rather than a “test” of their intelligence
- are active
- begin with what they understand
- draw on other information and sources of knowledge
- refer to the problem statement frequently
- employ sequential analysis when a question is initially unclear
- ask themselves questions and think out loud – (we call this "self talk")
- brainstorm
- use physical aids to thinking, such as drawing pictures
- relate problems to familiar / concrete experiences or to previous problems
- carefully proceed through a series of steps to reach a conclusion
Less successful problem solvers often:

- lack necessary subject-matter knowledge and are willing to allow gaps of knowledge to exist
- have a poor attitude: an attitude of indifference towards achieving a complete, accurate comprehension of situations and relations
- demonstrate a lack of objectivity in dealing with problems
- are passive in their thinking
- have great difficulty in ascertaining what they are required to do
- grope blindly toward a solution
- fail to complete a chain of reasoning
- use one-shot thinking, rather than building an extended, sequential path to understanding
- rush through or skip instructions
- take little time to consider a question or break it down into its component parts
- are mentally careless and superficial
- select an answer on the basis of a feeling or guess

Problem solving skills can be learned and improved. Study the characteristics here and identify both the skills you possess and the ones on which you need to work. As you use the techniques of a successful problem solver, you gradually become one.

II. ALGORITHMS

We’ve explained that programming consists of problem solving. Now we’ll go one step further, and tell you that programming uses a special form of problem solving that requires algorithmic thinking.

A. Algorithms In General

There are many new words that will be used in this course. Most of these words represent concepts with which you are already familiar. Algorithm is one of those words. An algorithm is a step-by-step set of instructions to accomplish a task. It is likely that you are unfamiliar with the word algorithm, but even so, you are familiar with algorithms themselves.

Think of a set of instructions with which you are familiar: the directions to Logan’s new Dance Club, a recipe for Bodacious Brownies, or the instructions to assemble a bike. These are all algorithms because they consist of a step-by-step set of instructions to accomplish a task.

You’ll even find an algorithm on your shampoo bottle! For example,

- lather
- rinse
- repeat

This particular algorithm leaves out many details. We might improve on this set of instructions by adding more detail:

- Wet hair
- Pick up shampoo bottle
- Open the bottle
Pour some shampoo on your hand
Close the shampoo bottle
Set bottle down
Put shampoo on hair
Rub shampoo through hair until lather covers hair
Rinse hair until all lather is removed
Repeat the previous eight steps one more time

There are still more details that could be added: for example, how much shampoo to put on hand or what temperature water should be. Depending for whom the instructions are written, this may be enough detail to get the task accomplished. Even with the original, three-step algorithm the task is often accomplished successfully because the user has enough experience to fill in the details.

Notice that this algorithm is written in an abbreviated form of English. These are not complete sentences. It is simply a series of steps. This could be called a “pseudo” language. It is not English in its true form, but is similar and understandable to someone who knows the English language.

There are three basic types of instructions, sequential, selection, and repetition. These different types of instructions control the order that operations are executed. Sequential statements are performed in sequential order, one after another. Selection statements control whether certain operations will or will not be executed depending on a condition. Repetition statements repeat a set of instructions over and over again. In computer science these are called control structures and we will use all three types.

**B. Algorithms in Computer Science**

So far, we have looked at a very basic definition of an algorithm: a step-by-step set of instructions to accomplish a task. In computer science, we have five very specific requirements of an algorithm:

1. **THE INSTRUCTIONS MUST BE WELL ORDERED.**

   This is crucial to the successful completion of the task. Consider the following hair washing algorithm:

   Rub shampoo through hair until lather covers hair
   Wet hair
   Pick up shampoo bottle
   Open the bottle
   Pour some shampoo on your hand
   Put lid back on the shampoo bottle
   Set bottle down
   Put shampoo on hair
   Rinse shampoo until hair is removed
   Repeat the previous steps once
Notice that the order of one step has been changed. The step **Rub shampoo through hair until lather covers hair** may seem like the most important step in the process of washing hair. However, putting this step first causes the entire procedure to be incorrect. When writing an algorithm, each instruction must be in the correct order.

2. **THE ALGORITHM MUST BE UNAMBIGUOUS, THAT IS, CLEAR AND INCAPABLE OF BEING MISUNDERSTOOD.**

Each step must be clearly defined and organized in a way that there is no doubt how to proceed. Consider the following instruction:

- Pour some shampoo in your hand or get another bottle

This is ambiguous. How is one to determine which thing to do? The instructions must be clear in specifying what is to be done in any situation. The above choice could be written as:

- If the bottle has shampoo in it, pour some in your hand; otherwise, get another bottle.

This instruction makes it clear what to do when the bottle has shampoo in it and what to do otherwise. It covers all cases and leaves no doubt how to proceed.

Another example of ambiguity is found in of simple set of shampoo bottle instructions we saw previously.

- lather
- rinse
- repeat

The ambiguity comes in the last statement, **repeat**. What is to be repeated? Does the person following the instructions repeatedly lather and rinse or just repeat the rinse? To clear this up, one might write:

- lather
- rinse
- repeat the previous two steps

For a set of instructions to be an algorithm, there can be no ambiguity.

3. **THE PROCESS MUST EVENTUALLY END.**

It is possible to write a set of instructions that, when followed exactly, would never end. For example, look our shampoo instructions again.

- lather
- rinse
- repeat the previous two steps

They never tell us when to stop repeating lather and rinse. If you followed the instructions exactly, you would continue washing your hair indefinitely … probably until you ran out of hot water… or hair!

When a set of instructions is to be repeated over and over, it is always necessary to provide a mechanism that causes the repetition is to stop.
We can rewrite the previous instructions as follows, so they meet this requirement for an algorithm:

```
lather
rinse
repeat the previous two steps one more time
```

4. **THE REQUESTED ACTION MUST BE "DOABLE."**

Since we are developing computer programs, the steps in the algorithm must be doable by a computer. There are some procedures that a computer cannot do, for example, dividing by zero. Mathematics teaches us that dividing by zero is undefined. Since the result is undefined, it is something that cannot be accomplished by a computer. It will be necessary to make sure such impossibilities not appear in our work.

5. **THE ALGORITHM MUST PRODUCE A RESULT.**

No matter how complex or elegant a solution is, if it doesn’t produce a result, it is of no value. Notice we did not say the algorithm must produce a solution, only a result. Sometimes the result is to identify a solution is not possible with the current conditions. Each algorithm, when followed, must produce a result. Often this result is in the form of output to the screen or to a file.

III. PROGRAM DEVELOPMENT (see also section 1.6 in your text)

The programming process can be broken down into a series of steps.

**A. Step 1 - Analyze the problem**

The purpose of this step is to understand the problem. This is a critical beginning. Without a sound understanding of the problem, a correct solution is not likely. Start by reading the problem statement. Then reread it and break it down into the following parts:

- **Input**
  Identify what information the program will have to retrieve. It may be information that the user will provide or some data that is available in a file.

- **Values given or known values**
  Identify any values that appear in the problem statement. These may or may not be used in your solution. Also watch for values or formulas that are not given but that will need to be used to complete the solution.

- **Processing**
  What will need to be done to produce the desired output? It is important at this stage to focus on what needs to be done, not how it will be accomplished.

- **Output**
  Identify what is to be produced by this program. This will be a list of the desired results.
Read through the problem statement and identify these different elements. Sometimes the problem statement will not clearly identify each of the four components. So first focus on the output to determine the result the program is to produce. Have a clear understanding of the desired outcome of the program. This will help to identify what input is needed and what process(es) must be accomplished by the program. You may find it helpful to underline or circle the different requirements or set them up in a table, as shown in the following example.

Consider the problem statement:
Write a program that gets the height, width, and depth of a box from the user and reports the volume of the box.

Starting with output, we can identify that the volume of the box is the required result of the program. List this under output. Then look for what will be supplied to the program, the input. The problem states the height, width, and depth of the box will be provided by a user. List these three items under input. There are no values given in the problem statement. The process to be performed is the calculation of volume. Put this under the process category. Note that we are not yet concerned with how to calculate the volume, only that it must be done.

<table>
<thead>
<tr>
<th>Input</th>
<th>Given Values</th>
<th>Processing</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>height</td>
<td>none</td>
<td>calculate volume</td>
<td>volume of box</td>
</tr>
<tr>
<td>width</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>depth</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Now reread the problem to see if you understand what is to be accomplished. You should only proceed to the next step in the programming process when you have a sound understanding of the problem. If you are not comfortable with your understanding of the problem statement seek help, ask questions. The instructor and the tutors are available to answer questions and clarify details. It is much better to get a clear view of the problem at this stage than to proceed with a gap in your understanding. Often, those gaps, if left unattended, will require additional time and unnecessary effort, and will likely produce errors later in the process.

**B. Step 2 - Outline the solution**

Once you have clearly defined the problem, it is time to organize the solution. Write down the major steps that need to be accomplished as a simple outline. Try to break the problem up into smaller parts. This technique is called *top-down design*.

In the previous step you identified the tasks that need to be accomplished; in this step, you begin to organize those tasks. Consider whether any steps need to be repeated or if there are alternate steps to be followed, for example what to do if there is no warm water. List the steps and establish the order in which they must be performed. Outline them in the correct order. Note that we are still not concerned with the processing details, only with what has to be done. Of course, for simple problems you may immediately recognize how to do the processing. But, the problems you will be solving will increase in difficulty and/or complexity. If you follow the design process as it is being described, you will be able to produce an accurate outline of the program without having to worry about the details. Think of the operation as writing an outline for an assistant who will perform the details to accomplish the broad tasks you specify. (Of
course, you will eventually have to specify those details; but for now the focus is designing the solution.)

Example:
For the problem statement analyzed in Step 1:
   Write a program that gets the height, width, and depth of a box from the user and reports the volume of the box.

There are no steps to be repeated, there are no alternate steps to follow, and the order is straightforward. Our simple English outline would look like:
   Get height, width and depth of the box from user
   Calculate the volume of the box
   Output the volume of the box

As your problem solving ability and programming expertise increase, you may find that you can combine this step with the next one—developing the algorithm.

**C. Step 3 - Develop an algorithm**

Since our ultimate goal is to write a series of instructions that can be carried out by a computer, we have to think about what needs to be done in terms of what a computer is able to do. We are not yet ready to write the instructions in a programming language. At this stage we are designing the solution. As we develop the design, we will keep in mind that we will be solving the problem with a computer program. The steps we develop will be steps that can be carried out by a computer.

Remember an algorithm is a step-by-step set of instructions to accomplish a task. In this step you simply write this list of instructions.

It is during this step that we add many of the details. What will our interaction with the user be? What formulas will we need to make the calculations?

For the problem statement analyzed in Step 1 and Step 2:
   Write a program that gets the height, width, and depth of a box from the user and reports the volume of the box.

The algorithm would look like:
   Print “Enter the height of the box: “
   Get height from user
   Print “Enter the width of the box: “
   Get width from user
   Print “Enter the depth of the box: “
   Get depth from user
   volume = height * width * depth
   Print “The volume of the box is “
   Print volume

Notice how there are many more details at this stage than there has been previously. We will discuss these details in section IV.
D. Step 4 - Test the algorithm

This step seeks to expose any logic errors in the algorithm. If these errors can be discovered at this point in the development, they can be corrected easily. If left unchecked, these errors take a great deal of time to discover and correct at the coding stage. Testing the algorithm consists of selecting sample input and then walking through the program one step at a time. It is recommended that you select two or three sets of sample data.

Basically, you will pretend you are the computer, following the instructions as the computer would, one at a time. This is called hand tracing. It is important that, during this process, you do what the instructions tell you to do, not what you think you should do. Otherwise, you may have errors in your algorithms that you don't detect.

When selecting sample input, select data that fall within the values specified in the problem statement. This would be another good time to reread the problem statement.

For Example:

Let’s use the following input to test the algorithm developed in Step 3:
- height – 3
- width – 5
- depth – 4

<table>
<thead>
<tr>
<th>Print “Enter the height of the box: “</th>
<th>Enter the height of the box: 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get height from user</td>
<td>height</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Print “Enter the width of the box: “</td>
<td>Enter the width of the box: 5</td>
</tr>
<tr>
<td>Get width from user</td>
<td>height width</td>
</tr>
<tr>
<td></td>
<td>3 5</td>
</tr>
<tr>
<td>Print “Enter the depth of the box: “</td>
<td>Enter the depth of the box: 4</td>
</tr>
<tr>
<td>Get depth from user</td>
<td>height width depth</td>
</tr>
<tr>
<td></td>
<td>3 5 4</td>
</tr>
<tr>
<td>volume = height * width * depth</td>
<td>volume = 3 * 5 * 4</td>
</tr>
<tr>
<td></td>
<td>volume</td>
</tr>
<tr>
<td></td>
<td>height width depth volume</td>
</tr>
<tr>
<td></td>
<td>3 5 4 60</td>
</tr>
<tr>
<td>Print “The volume of the box is “</td>
<td>The volume of the box is</td>
</tr>
<tr>
<td>Print volume</td>
<td>60</td>
</tr>
</tbody>
</table>
E. Step 5 - Code The Algorithm Into A Specific Language

This step brings us to the process of coding in a language that the computer can understand. The computer actually only understands machine language, a language consisting of 1s and 0s. This is a difficult language for people to work with. High-level programming languages like C++ have been developed to provide a language that is easier for people to understand and work with. They follow exact syntax rules. The exactness of the language makes it possible for tools such as a compiler to translate the programming language that uses keywords like if and while into the machine’s language of 0s and 1s.

In this course, we will be using the programming language C++. This is a powerful language and there are many details to learn. We will use the text, Starting Out with C++, 4th edition by Tony Gaddis to learn about the details of the C++ language.

F. Step 6 - Run and test the program on the computer

This is the most rewarding step of all. As you use a computer to execute the code you have designed and written you get to see first hand the results of your labors. You will be informed of any syntax errors you have made. Luckily, these can be easily corrected and the program can be rerun. And since you have taken the time to carefully and thoughtfully develop and test your algorithm, you should have, if not a final product, then one very close to final. You will have the satisfaction of seeing that you can write a complete solution that causes a computer to produce the desired results. Not bad!

G. Step 7 - Document and maintain the program

There are different types of documentation, including documentation designed for the user. However, in this course we will only be concerned with internal documentation that is designed for programmers. The documentation should make it easy for another programmer to understand your program and revise it, if necessary. (It will also help you remember what you did when you look at the program 2 weeks or six months from now!) We will learn to add comments that are ignored by the compiler, but which clarify details and explain the purposes and methods of our programs.

Maintenance involves keeping programs updated and secure. It is not a topic specifically covered in this course.

This course reader covers the first 4 steps. This process of analyzing the program, outlining the solution, developing an algorithm, and testing the algorithm is the core of good program development. When these steps are done effectively, better solutions are produced in less time.

IV. PSEUDOCODE FOR BASIC PROGRAMMING ACTIVITIES

Algorithms for computers can be expressed in many different ways, or if you will, in different “languages”. Ultimately we want an algorithm expressed in a specific programming language. However, programming languages have very rigid rules dealing with the way we can write an instruction. These grammar rules are called the syntax of the language. It is very difficult, error-prone, and time-consuming to try to write a program of any size or complexity directly in a
programming language. The programmer tends to get bogged down in the details of the language and loses sight of the more important aspects of solving the problem.

Several “algorithmic languages” have been developed to help the programmer first solve the problem. These include

- Simple English-like outlines as we have seen in our previous examples
- Pictorial representations, using special symbols to represent the various kinds of statements and lines with arrows to indicate the order of execution. These are called flowcharts.
- IPO diagrams and structure charts describe algorithms in chart form
- Pseudocode. This is the one we will use. An algorithm written in a programming language is referred to as code. Pseudo- or “false” code resembles code, but does not have rigid syntax rules, since it is meant to be interpreted by people, not computers

Our pseudocode language will include the following elements:

- **keywords**: words that have a special meaning
- **variables**: a place to store a value. A variable can store a single value. That value can change.
- **operators**: used to indicate that an operation is to be performed
  - for example, * to indicate multiplication
  - < to compare one value to another
- **punctuation**: the only punctuation we use are quotation marks, which are used to indicate the beginning and end of a string of text, and commas, used to separate items in a list.

### A. Key Words

We will be using the following key words. Each word has a specific function. We will discuss how they work when we first use them.

- Get
- Print
- And
- Or
- If
- Else
- End If
- Repeat
- End Repeat
- While
- End While

### B. Variables  (see also sections 1.4 and 1.5 in your text.)

The computer must have access to the values it needs to perform computations and to make comparisons.

A computer’s memory consists of millions of storage locations, each of which has a unique numeric *address*. The value stored in a location is referred to as the *contents* of that location.
When a value is placed in a storage location (we say it is written to the location), the new value replaces whatever value was previously located there. This is a destructive write. Thus, a location can hold only one value at a time and there is no way of retrieving any previous value. This is a very important concept.

When a value is retrieved (read from) from a storage location, the value is actually copied; the value remains in the storage location. This is a nondestructive read.

To make it easier to write instructions that tell the computer to read/write values from/to memory, we can assign names to storage locations so that, for example, we can talk about comparing the HoursWorked to the MinimumHours instead of having to talk about comparing the value stored in memory location 12345678 to the value stored in memory location 13184433.

A named storage location in computer memory is called a variable, since contents of the storage location can change (vary) during execution of a program. Note that the contents may change, but the address does not.

The computer can store different types of numeric data as well as character information, and each are treated differently. Many programming languages require that we define the type of data that will be stored as well as the variable name we are using. However, we will not require this as part of our algorithm development. For more on variables, see pages 14-15 in the text.

To use variables we need to decide on a name for them. The name must be unique and it must be a single word. For example:

- volume
- net_wages
- height
- sum
- salesTax
- InterestRate

Selecting the name of a variable is an important step. Notice how the variable names listed above describe the value they contain. Choosing a descriptive name can significantly improve the readability of the code, making it easier to develop and maintain. **We require the use of descriptive variable names in this course.**

**C. Operators**

When writing algorithms you may use basic mathematical operators to express the calculations to be done. There is one operator that has a very specific use in computer science that is different than how you have used it in math. It is the assignment operator: =

This operator looks like the equality operator used in algebra but it performs a different operation. This operator takes the value on the right side and stores it in the location specified on the left side.

The value on the right side can be a number, a variable, or an expression that produces a value. The left side must be a place that a value can be stored – a variable.
Examples:
result = 365 * 92 + 12

area = length * width

age=92

Note: Order is important here. The assignment operator works from right to left only. The following is not correct:
length * width = area

D. Control Structures

The default operation of most computers is to execute program instructions sequentially, one after another. However, it is often necessary to have the computer choose between two sets of instructions. For example, the computer may need to execute one set of instructions for calculating pay if the employee being processed is salaried and a different set if the employee is on commission.

Also, since a major advantage of using a computer to solve a problem is its ability to quickly and accurately perform the same calculations with different data, we often want to set up a series of instructions and have the computer perform them repeatedly. For example, we might want to calculate pay for all 48 employees in the firm. The way we do this is to set up the instructions on how to calculate the pay and tell the computer to do it 43 times. The beauty of this is that it would be just as easy to do it for 798 employees!

So, we see that we need to be able to control the order in which instructions will be performed, or as we say in programming, executed.

Computer scientists have proven mathematically that any problem that is solvable by computer can be solved using, at most, only three different kinds of control structures.

They are:

1. SEQUENCE
As mentioned, this is the “native” mode of execution. In a sequence structure, the instructions are completed in series, one right after another. Most of the hair washing hair algorithm is this type, for example, pick up shampoo bottle, take lid off bottle.

2. SELECTION (sometimes called decision)
This structure is used to select which steps will be executed, depending on whether a certain condition is true or false. We are all familiar with the concept of “if”. We saw an if in the hair-washing algorithm with respect to the availability of warm water. The step “get another bottle of shampoo”, will only be performed if the condition, “the bottle is empty”, is true.

3. REPETITION (also referred to as looping or iteration)
In a repetition structure, a set of instructions are repeated. “Repeat the previous two steps once” sets up a repetition structure. In this case it instructs the "shampooer" (processor) to perform a
series of steps one more time. Repetition structures may be set up so that a group of instructions is repeated a certain number of times or until a condition changes, for example, “While there are more employees”…(set of instruction to be repeated).

E. Producing Output

A requirement of an algorithm is that it produce a result. Often this result is some type of output. This output may be displayed on a monitor or written to a file. When we want to produce output we will use the keyword Print.

Output may take different forms, for example: a message to the user; the value of a variable; the result of a computation. When we wish to indicate that a series of specific characters are to be output, we will enclose the characters in quotation marks, “”. (A series of characters is referred to as a string.) When we want to output the value of a variable, we simply refer to the variable by name.

Examples:
Print “Welcome!”
This command will display the string Welcome! Notice that the quotation marks are used to identify the beginning and end of the string and are not actually displayed.

Print volume
This command will display the current value of the variable volume. When referring to variables use only the name of the variable (no quotation marks.)

Print height * width
This command will first do the calculation height * width and then display the result.

When displaying output it is important to label it. Labeling output is simply displaying information that explains the values being output. For example:

Print “The volume of the box is “
Print volume

This example uses two Print commands. The first is just the label and the second is the actual value that is to be output. It is possible to combine multiple outputs in a single Print command. To Print multiple outputs, simply separate the outputs with a comma. The comma will not be output. The following single Print command will produce the same output as the 2 Print commands above.

Print “The volume of the box is “, volume

F. Getting Input

When our problem requires the computer to retrieve values and put them in storage locations, we will use the keyword Get. Get must be followed by the name of the variable where the value is to be stored. You can specify where you are retrieving the information from by adding the words from user or from file at the end of the Get statement.
Example:
Get height from user

This statement will get one value from a user and store that value in the variable height. When getting values from a user, the Get command needs to be preceded by a prompt. A prompt is a Print statement that tells the user what you want them to enter. To do the preceding example correctly it would look like this:

Print “Enter the height: “
Get height from user

It is possible to Get more than one value in a single Get statement by listing more than one variable name each separated by a comma.

Print “Enter the length and width of the rectangle: “
Get length, width from user

This statement will accept two values from the user. The first will be stored in the variable length and the second will be stored in the variable width.

When reading from a file, input prompts are not needed.

Get length, width from file

**G. Sample Algorithm using sequence structures**

Let's try a problem with what we know so far.

Write an algorithm for a company that sells three types of bags: small, medium, and large. They sell for $2.50, $3.25, and $4.00 respectively. This algorithm is to read three numbers from a file. The first number indicates how many small bags the customer has ordered. The second number indicates how many medium bags the customer has ordered. And the last number indicates how many large bags the customer has ordered. The algorithm will calculate and display to the screen the total cost of the bags, the sales tax charged at a rate of 7.5%, and the total sale including sales tax.

Take a minute to look at how you reacted to that problem statement. It has a considerably higher level of complexity. Did you find your eyes crossing and your vision fading as you read it? Did your hands get sweaty and your stomach turn over? If not, consider yourself fortunate! If so, what are you going to do about it?

The first thing to realize is that most people suffer some level of anxiety when faced with a new problem. That includes your instructors! This anxiety can seriously influence a person's ability to solve the problem. It is important to recognize that this is an emotional response and does not reflect one's intellectual ability.

If you suffer from such anxiety, here are a couple of more ideas that may help.

- Take a couple of deep breaths. This may calm you down and allow you to use a more thoughtful approach.
• Use "self talk". Tell yourself that you CAN do this. If you believe you can do something, it is much more likely that you will be able to do it.
• Ask yourself what a successful problem solver might do.
• Start by identifying what you do understand.
• Start with the first step:
  o look for what the program is to output. Working on this step, you will realize that you are quite capable of accomplishing this task. (In fact you may find that it is easy!)
  o look for the input. You will also be able to do this.
  o identify the given values
  o decide what processing needs to be done to get the desired output
• What you will find is that you can do each step, one at a time. This will increase your confidence in your ability to complete the entire problem.

Back to our problem—

Step 1 - Analyze the problem

a. Identify what we are looking for, the output:
   total cost of bags
   sales tax
   total sale

b. Look for the input:
   There will be three numbers in a file. They are
   quantity of small bags ordered
   quantity of medium bags ordered
   quantity of large bags ordered

c. Look for Given Values
   This problem statement gives us several values that are needed to do the calculations.
   We will want to make a note of these values.
   price of a small bag - $2.50
   price of a medium bag - $3.25
   price of a large bag - $4.00
   the sales tax rate – 7.5%

d. What processing will need to be done?
   We will need to
   calculate total cost
   calculate the amount of sales tax
   calculate the total sale

You may find it helpful to put these in lists so the information can be easily seen:

<table>
<thead>
<tr>
<th>Input</th>
<th>Values Given</th>
<th>Processing</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>quantity of small bags</td>
<td>price of small $2.50</td>
<td>calculate total cost</td>
<td>total cost of bags</td>
</tr>
</tbody>
</table>
quantity of medium | price of medium $3.25 | calculate sales tax | sales tax
quantity of large bags | price of large $4.00 | calculate total | total sale
sales tax rate 7.5%

After extracting all this information from the problem statement, read the problem again to make sure you understand what it is asking for.

**Step 2: Outline the solution.**

Make a list of what needs to be done. It is critical at this point to make sure the steps are in the correct order. In this problem there are no choices to be made and the computations are only done once, so it can all be done with sequence structures.

- Set the initial prices of the bags
- Set the value of the tax rate
- get the numbers from the file
- calculate cost
- calculate sales tax
- calculate total
- print the results to the screen

**Step 3: Develop an algorithm**

In this step fill out the outline by adding the details. Notice we use variables to store all the values even though they don’t change. See how we wrote 7.5%. Also, the prices could have been written different. The zeros at the end do not change the values so we could have used 2.5, 3.25, and 4 and produced the same result.

```python
small_price = 2.50
medium_price = 3.25
large_price = 4.00
tax_rate = .075
Get smallQty, mediumQty, largeQty
cost = small_price * smallQty + medium_price * mediumQty + large_price * largeQty
sales_tax = cost * tax_rate
total = cost + sales_tax
Print "The cost of the bags is ", cost
Print "The sales tax is ", sales_tax
Print "The total sale is ", total
```

**Step 4: Test the algorithm**

Go through the algorithm one step at a time doing just what each instruction says. Track the variables as you go. We will use 5, 8, and 10 as the input values. It is a good idea to have in mind what you expect the results to be.

<table>
<thead>
<tr>
<th>small_price = 2.50</th>
<th>small_price</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.50</td>
<td>2.50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>medium_price = 3.25</th>
<th>small_price</th>
<th>medium_price</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.50</td>
<td>2.50</td>
<td>3.25</td>
</tr>
</tbody>
</table>
large_price = 4.00

<table>
<thead>
<tr>
<th>small_price</th>
<th>medium_price</th>
<th>large_price</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.50</td>
<td>3.25</td>
<td>4.00</td>
</tr>
</tbody>
</table>

tax_rate = .075

<table>
<thead>
<tr>
<th>small_price</th>
<th>medium_price</th>
<th>large_price</th>
<th>tax_rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.50</td>
<td>3.25</td>
<td>4.00</td>
<td>.075</td>
</tr>
</tbody>
</table>

Get smallQty, mediumQty, largeQty

<table>
<thead>
<tr>
<th>small_price</th>
<th>medium_price</th>
<th>large_price</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.50</td>
<td>3.25</td>
<td>4.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>smallQty</th>
<th>mediumQty</th>
<th>largeQty</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>10</td>
</tr>
</tbody>
</table>

cost = small_price * smallQty + medium_price * mediumQty + large_price * largeQty

<table>
<thead>
<tr>
<th>small_price</th>
<th>medium_price</th>
<th>large_price</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.50</td>
<td>3.25</td>
<td>4.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>smallQty</th>
<th>mediumQty</th>
<th>largeQty</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>10</td>
<td>78.50</td>
</tr>
</tbody>
</table>

sales_tax = cost * tax_rate

<table>
<thead>
<tr>
<th>small_price</th>
<th>medium_price</th>
<th>large_price</th>
<th>tax_rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.50</td>
<td>3.25</td>
<td>4.00</td>
<td>.075</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>smallQty</th>
<th>mediumQty</th>
<th>largeQty</th>
<th>cost</th>
<th>sales_tax</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>10</td>
<td>78.50</td>
<td>5.8875</td>
</tr>
</tbody>
</table>

total = cost + sales_tax

<table>
<thead>
<tr>
<th>small_price</th>
<th>medium_price</th>
<th>large_price</th>
<th>tax_rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.50</td>
<td>3.25</td>
<td>4.00</td>
<td>.075</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>smallQty</th>
<th>mediumQty</th>
<th>largeQty</th>
<th>cost</th>
<th>sales_tax</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>10</td>
<td>78.50</td>
<td>5.8875</td>
<td>84.3875</td>
</tr>
</tbody>
</table>

Print "The cost of the bags is ", cost

The cost of the bags is 78.50

Print "The sales tax is ", sales_tax

The sales tax is 5.8875

Print "The total sale is ", total

The total sale is 84.3875

Are these the results we expect? These values look correct. There is one problem though; the sales tax and total sale values are not in dollars and cents. This could create problems if your output is expected to be in dollar and cents. The problem arises when we multiply by .075. Let's add an additional step that resolves the problem:

small_price = 2.50
medium_price = 3.25
large_price = 4.00
tax_rate = .075
Get smallQty, mediumQty, largeQty
cost = small_price * smallQty + medium_price * mediumQty + large_price * largeQty
sales_tax = cost * tax_rate
round sales_tax to 2 decimal places
total = cost + sales_tax
Print "The cost of the bags is ", cost
Print "The sales tax is ", sales_tax
Print "The total sale is ", total

small_price = 2.50

<table>
<thead>
<tr>
<th>small_price</th>
<th>medium_price</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.50</td>
<td>3.25</td>
</tr>
<tr>
<td>large_price = 4.00</td>
<td>small_price</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------</td>
</tr>
<tr>
<td></td>
<td>2.50</td>
</tr>
<tr>
<td>tax_rate = .075</td>
<td>small_price</td>
</tr>
<tr>
<td></td>
<td>2.50</td>
</tr>
<tr>
<td>Get smallQty, mediumQty, largeQty</td>
<td>small_price</td>
</tr>
<tr>
<td></td>
<td>2.50</td>
</tr>
<tr>
<td></td>
<td>smallQty</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td>cost = small_price * smallQty + medium_price * mediumQty + large_price * largeQty</td>
<td>small_price</td>
</tr>
<tr>
<td></td>
<td>2.50</td>
</tr>
<tr>
<td></td>
<td>smallQty</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td>sales_tax = cost * tax_rate</td>
<td>small_price</td>
</tr>
<tr>
<td></td>
<td>2.50</td>
</tr>
<tr>
<td></td>
<td>smallQty</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td>round sales_tax to 2 decimal places</td>
<td>small_price</td>
</tr>
<tr>
<td></td>
<td>2.50</td>
</tr>
<tr>
<td></td>
<td>smallQty</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td>total = cost + sales_tax</td>
<td>small_price</td>
</tr>
<tr>
<td></td>
<td>2.50</td>
</tr>
<tr>
<td></td>
<td>smallQty</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

Print "The cost of the bags is ", cost
The cost of the bags is 78.50
Print "The sales tax is ", sales_tax
The sales tax is 5.89
Print "The total sale is ", total
The total sale is 84.39

There, now we have a better solution.

**H. Conditions**

A condition is a question whose answer is either true or false. (A value that can take on one of only two possible values is called a *Boolean* value, after the mathematician George Boole) We will use these questions as we develop selection and repetition structures.

Often we ask these kinds of questions by using comparison operators such as the following:

- `<`  
  *Less Than*

- `>`  
  *Greater Than*

- `==`  
  *Equal To* (the double equal sign is used to distinguish it from the = operator, which assigns a value to a variable)

- `! =`  
  *Not Equal To*

- `< =`  
  *Less Than or Equal To*

- `> =`  
  *Greater Than or Equal To*
Examples:
elevation = 10000
Question: Is the value stored in \texttt{elevation} equal to 10000? This condition will be true only when the value in \texttt{elevation} is 10000. (Note that we do not use commas in large numbers.)

size < 12
Question: Is the value stored in \texttt{size} less than 12? This condition will be true when the value stored in \texttt{size} is less than 12. Note that the condition will be false when the value in \texttt{size} is exactly 12.

quantity >= 100
Question: Is the value stored in \texttt{quantity} greater than or equal to 100? This condition will be true when the value stored in \texttt{quantity} is greater than 100 and will also be true when the value stored in \texttt{quantity} is equal to 100. It will only be false when the value stored in \texttt{quantity} is less than 100.

92.5 > cost
Question: Is 92.5 greater than the value stored in cost? This condition will be true when the value stored in cost is greater than 92.5. Note the condition will be false when the value in cost is 92.5.

volume != 78.9
Question: Is the value stored in volume not equal to 78.9? This comparison is sometimes confusing to read, but if you just take a moment to think about what you are asking, you should be OK. This condition will be true when the value stored in volume is not 78.9. It will be false only when the value stored in volume is 78.9. You're probably wondering why we would want to ask a question in this way. You will discover that we frequently want to repeat a series of instructions as long as the user does not enter some specific value or that we only want to execute a series of steps when a certain situation does not exist. Hang in there, and it will make more sense when you have more experience.

We can create more complex conditions by using the keywords And and Or. These key words are used to combine two or more conditions. When you want to compose a complex condition, first decide on the questions you want to ask. Each condition will be a complete question. Then decide whether you want to combine the questions with And or Or. Each produces different results. The And requires that both conditions are true for the entire condition to be true. The Or only requires that one of the conditions be true for the entire condition to be true.

Examples:
temperature > 32 And temperature < 212
This complex condition will be true only when the value in temperature is in the range 32-212, exclusive that is, not including 32 and 212.

temperature <=32 Or temperature >=212
This complex condition will be true for all values of temperature outside the range of 32-212, inclusive, that is, for values greater than or equal to 212, as well as values of temperature less than or equal to 32.

\[ \text{purchase} > 250 \text{ Or } \text{isEmployee} = = \text{true} \]
(Note that true is a Boolean value; that means \text{isEmployee} must be a Boolean variable) This complex condition will be true when purchase is greater than 250, as well as when the value in \text{isEmployee} is true. The only time this condition is false is when purchase is less than or equal to 250 at the same time the value in \text{isEmployee} is not true.

\[ \text{temperature} > 32 \text{ Or } \text{temperature} < 212 \]
This complex condition will be true for any value stored in temperature. (Can you explain why?)

I. The If Statement

The selection control structure is used when different actions are to occur depending on a certain condition. The computer can examine a condition, determine whether the condition is true or false, and proceed based on that condition. The statement used to implement such a control structure in pseudocode is the if statement. It uses the keywords If, Else, and End If.

Example:

If age < 18
  Print "You cannot vote yet."
Else
  Print "You can vote, remember to register by September 15th."
End If

Only one of the print statements will be executed depending on the value of age. The condition, age < 18 is tested. When the value in age is less than 18, the condition is true and \text{You cannot vote yet} will be printed to the monitor and control is passed to the statement following the End If. If the value in age is not less than 18 (if it is 18 or larger), the condition is false and \text{You can vote, remember to register by September 15th} will be printed. Control will pass to the statement following the End If. Note that only one of the Print statements can be executed. Also, notice that the statements inside the If/Else portions are indented. This enhances readability.

Look at the following selection structure:

If temperature > 32 And temperature < 212
  Print "H_2O is a liquid at this temperature."
Else
  Print "H_2O is not liquid at this temperature."
End If

Can you identify what will be printed when temperature has each of the following values?

\[ \text{temperature} = 0 \]
Now try this one:

\[
\begin{align*}
\text{If } & \text{ purchase } > 250 \text{ Or isEmployee } = \text{ true} \\
\text{discount } & = .15 \\
\text{Else} \\
\text{discount } & = .05 \\
\text{End If}
\end{align*}
\]

\[
\begin{align*}
\text{purchase } & = 300, \text{ isEmployee } = \text{ true} \\
\text{purchase } & = 200, \text{ isEmployee } = \text{ true} \\
\text{purchase } & = 300, \text{ isEmployee } = \text{ false} \\
\text{purchase } & = 200, \text{ isEmployee } = \text{ false} \\
\text{purchase } & = 250, \text{ isEmployee } = \text{ true} \\
\text{purchase } & = 250, \text{ isEmployee } = \text{ false}
\end{align*}
\]

There are times that a program needs to do something if a certain condition is true and nothing if it is false. To accomplish this, the Else portion can be left off.

\[
\begin{align*}
\text{If temperature } & < 55 \\
\text{Print } & \text{ "Remember to take your jacket"} \\
\text{End If}
\end{align*}
\]

It is possible to have more than one statement in the true and/or in the false portion of an If statement.

\[
\begin{align*}
\text{Print } & \text{ "Enter the type of shape you have, circle or rectangle: "} \\
\text{Get choice} \\
\text{If choice } & = \text{ "rectangle"} \\
\text{Print } & \text{ "Please enter the length and width of the rectangle: "} \\
\text{Get length, width} \\
\text{area } & = \text{ length } \times \text{ width} \\
\text{Else} \\
\text{Print } & \text{ "Please enter the radius of the circle: "} \\
\text{Get radius} \\
\text{area } & = \pi \times \text{ radius } \times \text{ radius} \\
\text{End If} \\
\text{Print } & \text{ "The area of your ", choice, " is ", area}
\end{align*}
\]

J. Sample Algorithm using selection structures

Let’s try a problem that uses selection structures.
Write a program that reads an employee's information from a user and calculates their pay for the week. The input includes the employee name, hours worked this week and their hourly pay rate. The employee is to be paid their basic wage for the first 40 hours worked and time and a half for all hours above 40. Output the name, regular pay, overtime pay, and total pay for the week to the screen. If the employee worked 40 hours or less, do not output any information about overtime pay.

Step 1 - Analyze the problem

<table>
<thead>
<tr>
<th>input</th>
<th>given values</th>
<th>processing</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>employee name</td>
<td>over 40 hours is overtime</td>
<td>calculate regular</td>
<td>employee name</td>
</tr>
<tr>
<td>hours worked</td>
<td>overtime is 1.5 of regular pay</td>
<td>calculate overtime</td>
<td>regular pay</td>
</tr>
<tr>
<td>pay rate</td>
<td></td>
<td>calculate total</td>
<td>overtime pay (only when</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pay</td>
<td>there is overtime)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>total pay</td>
</tr>
</tbody>
</table>

Step 2: Outline the solution.

In this step we list what needs to be done and define the order. The order will be to get input, calculate values needed, and then print output. This is a common order. In the outline, we will specify what is to be input, what calculations need to be done, and what amounts are to be output.

We also decide what structure(s) we will use. The input will be done one after another so the default sequence structure will be sufficient. Different calculations are needed depending on whether the employee had overtime or not. This indicates there are two options depending on a certain condition. A selection structure will work best for this task. The output will also vary depending on overtime so we will want to use another selection structure in the output section of this outline.

Get name, hours worked, and pay rate from a user
If there was overtime, calculate regular pay on the first 40 hours and 1½ pay for additional hours
Otherwise, calculate all hours worked at regular pay
Print employee name and regular pay,
If there was overtime, print the overtime pay
Print total pay

Step 3: Develop an algorithm

In this step we need to fill in the details. We will select variable names that are descriptive and are a single word. We will write a condition that will determine whether an employee has overtime or not. We will construct the formulas to do the calculations. We will add input prompts and output labels.

We also need to convert to pseudocode. This consists of using keywords and indentation as needed.
Print “Enter employee name: “
Get name
Print “Enter hours worked: “
Get hours
Print “Enter rate of pay: “
Get payRate
If hours > 40
    regularPay = 40 * payRate
    overtimePay = (hours – 40) * payRate * 1.5
Else
    regularPay = hours * payRate
    overtimePay = 0
End If
totalPay = regularPay + overtimePay
Print "Payroll for ", name
Print "Regular Pay: ", regularPay
If overtimePay != 0
    Print "Overtime Pay: ", overtimePay
End If
Print "Total for this week: ", totalPay

Step 4: Test the algorithm

Let’s test it first with the following input:
John
36
10

<table>
<thead>
<tr>
<th>Print “Enter employee name: “</th>
<th>Enter employee name: John</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get name</td>
<td>name</td>
</tr>
<tr>
<td></td>
<td>John</td>
</tr>
<tr>
<td>Print “Enter hours worked: “</td>
<td>Enter hours worked: 36</td>
</tr>
<tr>
<td>Get hours</td>
<td>name</td>
</tr>
<tr>
<td></td>
<td>hours</td>
</tr>
<tr>
<td></td>
<td>John</td>
</tr>
<tr>
<td>Print “Enter rate of pay: “</td>
<td>Enter rate of pay: 10</td>
</tr>
<tr>
<td>Get payRate</td>
<td>name</td>
</tr>
<tr>
<td></td>
<td>hours</td>
</tr>
<tr>
<td></td>
<td>payRate</td>
</tr>
<tr>
<td></td>
<td>John</td>
</tr>
<tr>
<td>If hours &gt; 40</td>
<td>this condition is false so we will go to the Else portion</td>
</tr>
<tr>
<td>regularPay = hours * payRate</td>
<td>name</td>
</tr>
<tr>
<td></td>
<td>hours</td>
</tr>
<tr>
<td></td>
<td>payRate</td>
</tr>
<tr>
<td></td>
<td>regularPay</td>
</tr>
<tr>
<td></td>
<td>360</td>
</tr>
<tr>
<td>overtimePay = 0</td>
<td>name</td>
</tr>
<tr>
<td></td>
<td>hours</td>
</tr>
<tr>
<td></td>
<td>payRate</td>
</tr>
<tr>
<td></td>
<td>regularPay</td>
</tr>
<tr>
<td></td>
<td>overtimePay</td>
</tr>
<tr>
<td>totalPay = regularPay + overtimePay</td>
<td>name</td>
</tr>
<tr>
<td></td>
<td>hours</td>
</tr>
<tr>
<td></td>
<td>payRate</td>
</tr>
<tr>
<td></td>
<td>regularPay</td>
</tr>
<tr>
<td></td>
<td>overtimePay</td>
</tr>
<tr>
<td></td>
<td>360</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

| Print "Payroll for ", name   | Payroll for John          |
| totalPay                     | 360                       |
Now let’s try it with different input:

Bill
47
8.25

Print “Enter employee name: “
Get name
name
Bill

Print “Enter hours worked: “
Get hours
name
hours
Bill
47

Print “Enter rate of pay: “
Get payRate
name
hours
payRate
Bill
47
8.25

If hours > 40
this condition is true so we will do the first portion
regularPay = 40 * payRate
name
hours
payRate
regularPay
Bill
47
8.25
330

overtimePay = (hours – 40) * payRate * 1.5
name
hours
payRate
regularPay
overtimePay
Bill
47
8.25
330
86.625

totalPay = regularPay + overtimePay
name
hours
payRate
regularPay
overtimePay
totalPay
Bill
47
8.25
330
86.625
416.625

Print "Payroll for ", name
Payroll for Bill

Print "Regular Pay: ", regularPay
Regular Pay: 330

If overtimePay != 0
this condition is true so we will do the first portion
Print "Overtime Pay: ", overtimePay
Overtime Pay: 86.625

Print "Total for this week: ", totalPay
Total for this week: 416.625

Again it looks like we would benefit from rounding the result of overtime pay. Let’s consider if there are other calculations that might produce more than two digits after the decimal place. It would be possible to have a combination that would produce more digits in the regular pay, for example: 48.5 hours and 5.75 pay rate would produce regular pay of 278.875. So we need to round regular pay. If regular pay and overtime pay are both rounded to two digits after the decimal place, when we add them to get total pay they will not have more than two digits so we won’t need to round total pay.

Print “Enter employee name: “
Get name
Print “Enter hours worked: “
Get hours
Print “Enter rate of pay: “
Get payRate
If hours > 40
    regularPay = 40 * payRate
    overtimePay = (hours – 40) * payRate * 1.5
Else
    regularPay = hours * payRate
    overtimePay = 0
End If

round regularPay
round overtimePay

totalPay = regularPay + overtimePay
Print "Payroll for ", name
Print "Regular Pay: ", regularPay
If overtimePay != 0
    Print "Overtime Pay: ", overtimePay
End If
Print "Total for this week: ", totalPay

Print “Enter employee name: “
Enter employee name: Bill

Get name
name
Bill

Print “Enter hours worked: “
Enter hours worked: 36

Get hours
name
hours
Bill
47

Print “Enter rate of pay: “
Enter rate of pay: 8.25

Get payRate
name
hours
payRate
Bill
47
8.25

If hours > 40
    this condition is true so we will do the first portion
    regularPay = 40 * payRate
    name
    hours
    payRate
    regularPay
    Bill
    47
    8.25
    330

    overtimePay = (hours – 40) * payRate * 1.5
    name
    hours
    payRate
    regularPay
    overtimePay
    Bill
    47
    8.25
    330
    86.625

round regularPay
round overtimePay

round regularPay
round overtimePay

totalPay = regularPay + overtimePay
name
hours
payRate
regularPay
overtimePay
Bill
47
8.25
330
86.63

Print "Payroll for ", name
Payroll for Bill

Print "Regular Pay: ", regularPay
Regular Pay: 330

If overtimePay != 0
    this condition is true so we will do the first portion
    Print "Overtime Pay: ", overtimePay
    Overtime Pay: 86.63

Print "Total for this week: ", totalPay
Total for this week: 416.63
K. Repetition Control Structures

A repetition structure is used when a series of instructions is to be repeated. The repeated instructions make up the body of the loop. There are two basic forms of the repetition structure, the counted loop and the condition-controlled loop.

1. THE COUNTED LOOP

The counted loop simply indicates where the repetition is to begin, where it is to end and how many times it is to be repeated. It uses the keywords Repeat and End Repeat. Statements to be repeated are indented.

Consider the following algorithm:

```
month = 1
total = 0
Repeat 5 Times
    Print "How much rainfall was recorded for month, ", month, "? "
    Get monthlyRainfall
    total = total + monthlyRainfall
    month = month + 1
End Repeat
average = total / 5
Print "The average rainfall for the 5 months was ", average
```

The first two statements, month = 1 and total = 0 will be executed one time each and then processing will move on to the Repeat structure. The four statements inside the repeat will be executed in order exactly 12 times. After the last time the loop repeats, processing will move on to the statement that follows the End Repeat.

Notice that this loop gets a value for monthlyRainfall each time through the loop. Since a variable can only hold a single value, each time through the loop a new value is stored in monthlyRainfall and the previous value is overwritten. When the repetition is complete monthlyRainfall will contain the last value entered. The variable month will also change each time through the loop.

Look closely at what is happening to total. Outside the loop it is initialized to zero. Then each time through the loop, the new value of monthlyRainfall is added to the current value of total and the result of that addition replaces the value stored in total. When the repetition is complete, total will contain the sum of all 12 monthlyRains entered. Total is an example of a variable called an accumulator. Accumulators are common in programming solutions.

Notice that the value of month is initialized to 1 and increased by 1 each time we execute the loop. In this example, month is a counter. A counter is a variable that is initialized to a beginning value and then is increased by one each time something is counted. When we increase the value of a variable by one each time through a loop, we say the variable is incremented. Similarly, if we decrease the value of a variable by a specific amount each time through a loop, we say the variable is decremented. Counting is another very common use of variables.

We can learn a lot by tracing this algorithm. We will use the following input:

```
1.73  .87  .52  .93  1.12
```
<table>
<thead>
<tr>
<th>month = 1</th>
<th>month</th>
</tr>
</thead>
<tbody>
<tr>
<td>total = 0</td>
<td>total</td>
</tr>
</tbody>
</table>

Repeat 5 Times

Print "How much rainfall was recorded for month ",month,"?"

<table>
<thead>
<tr>
<th>month</th>
<th>total</th>
<th>monthlyRainfall</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1.73</td>
</tr>
</tbody>
</table>

Get monthlyRainfall

<table>
<thead>
<tr>
<th>month</th>
<th>total</th>
<th>monthlyRainfall</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1.73</td>
</tr>
</tbody>
</table>

total = total + monthlyRainfall

<table>
<thead>
<tr>
<th>month</th>
<th>total</th>
<th>monthlyRainfall</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.73</td>
<td>1.73</td>
</tr>
</tbody>
</table>

month = month + 1

<table>
<thead>
<tr>
<th>month</th>
<th>total</th>
<th>monthlyRainfall</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.73</td>
<td>1.73</td>
</tr>
</tbody>
</table>

Print "How much rainfall was recorded for month ",month,"?"

<table>
<thead>
<tr>
<th>month</th>
<th>total</th>
<th>monthlyRainfall</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.73</td>
<td>.87</td>
</tr>
</tbody>
</table>

Get monthlyRainfall

<table>
<thead>
<tr>
<th>month</th>
<th>total</th>
<th>monthlyRainfall</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.73</td>
<td>.87</td>
</tr>
</tbody>
</table>

total = total + monthlyRainfall

<table>
<thead>
<tr>
<th>month</th>
<th>total</th>
<th>monthlyRainfall</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2.6</td>
<td>.87</td>
</tr>
</tbody>
</table>

month = month + 1

<table>
<thead>
<tr>
<th>month</th>
<th>total</th>
<th>monthlyRainfall</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2.6</td>
<td>.87</td>
</tr>
</tbody>
</table>

Print "How much rainfall was recorded for month ",month,"?"

<table>
<thead>
<tr>
<th>month</th>
<th>total</th>
<th>monthlyRainfall</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2.6</td>
<td>.52</td>
</tr>
</tbody>
</table>

Get monthlyRainfall

<table>
<thead>
<tr>
<th>month</th>
<th>total</th>
<th>monthlyRainfall</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2.6</td>
<td>.52</td>
</tr>
</tbody>
</table>

total = total + monthlyRainfall

<table>
<thead>
<tr>
<th>month</th>
<th>total</th>
<th>monthlyRainfall</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3.12</td>
<td>.52</td>
</tr>
</tbody>
</table>

month = month + 1

<table>
<thead>
<tr>
<th>month</th>
<th>total</th>
<th>monthlyRainfall</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3.12</td>
<td>.52</td>
</tr>
</tbody>
</table>

Print "How much rainfall was recorded for month ",month,"?"

<table>
<thead>
<tr>
<th>month</th>
<th>total</th>
<th>monthlyRainfall</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3.12</td>
<td>.93</td>
</tr>
</tbody>
</table>

Get monthlyRainfall

<table>
<thead>
<tr>
<th>month</th>
<th>total</th>
<th>monthlyRainfall</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3.12</td>
<td>.93</td>
</tr>
</tbody>
</table>

total = total + monthlyRainfall

<table>
<thead>
<tr>
<th>month</th>
<th>total</th>
<th>monthlyRainfall</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4.05</td>
<td>.93</td>
</tr>
</tbody>
</table>

month = month + 1

<table>
<thead>
<tr>
<th>month</th>
<th>total</th>
<th>monthlyRainfall</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4.05</td>
<td>.93</td>
</tr>
</tbody>
</table>

Print "How much rainfall was recorded for month ",month,"?"

<table>
<thead>
<tr>
<th>month</th>
<th>total</th>
<th>monthlyRainfall</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4.05</td>
<td>1.12</td>
</tr>
</tbody>
</table>

Get monthlyRainfall

<table>
<thead>
<tr>
<th>month</th>
<th>total</th>
<th>monthlyRainfall</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4.05</td>
<td>1.12</td>
</tr>
</tbody>
</table>

total = total + monthlyRainfall

<table>
<thead>
<tr>
<th>month</th>
<th>total</th>
<th>monthlyRainfall</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5.17</td>
<td>1.12</td>
</tr>
</tbody>
</table>

month = month + 1

<table>
<thead>
<tr>
<th>month</th>
<th>total</th>
<th>monthlyRainfall</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5.17</td>
<td>1.12</td>
</tr>
</tbody>
</table>

we have done the loop 5 times so we move on

average = total / 5

<table>
<thead>
<tr>
<th>month</th>
<th>total</th>
<th>monthlyRainfall</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5.17</td>
<td>1.12</td>
<td>1.034</td>
</tr>
</tbody>
</table>

Print "The average rainfall for the 5 months was ", average

The average rainfall for the 5 months was 1.034
2. SAMPLE ALGORITHM USING REPETITION STRUCTURES

Let's try developing the algorithm for a problem that uses a counted loop.

Write a program that will produce a multiplication table for a user. Ask the user which times table they want produced. Then display the multiplication table from 0 to 12 for the given number.

Sample output desired:
If the user enters 7, the program will produce the following:
The multiplication table for 7 is:
7 x 0 = 0
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
7 x 11 = 77
7 x 12 = 84
Goodbye

Step 1 - Analyze the problem

<table>
<thead>
<tr>
<th>input</th>
<th>given values</th>
<th>processing</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>the times table</td>
<td>table to go from 0 to 12</td>
<td>calculate the multiplication table for the given number</td>
<td>The multiplication table for the given number</td>
</tr>
</tbody>
</table>

Step 2: Outline the solution.

In this problem we see that we are going to calculate and print a single line of the multiplication table 13 times, one for each value from 0-12. The requirement of doing something over and over again indicates the need for a repetition structure. Since we know how many times we are going to do the calculating and printing, we will use a counted loop.

When using a repetition structure it is important to determine what instructions are to be done only once and which need to be repeated. This will help determine the order of the statements. There are two lines, each of which is to be printed only once; The multiplication table for 7 is: and Goodbye. Since each of these statements execute only once, they will be outside the loop. Statements to be repeated will be inside the loop.

We also notice that the number entered by the user (7 in the example above) stays constant each time through the loop but the value we multiply it by is incremented by 1 each time. We will need a variable that increments by one each time through the loop.
Get a value from the user
print The multiplication table for __ is:
loop 13 times, each time
calculate the result of user's number * multiplier
print the problem and the answer
end the loop
print goodbye

Step 3: Develop an algorithm

Now we need to add the details. We decide what variables will be needed and select descriptive, single word variable names. We add input prompts and output labels. We construct the formulas to do the calculations. Also, we initialize and increment the multiplier. Some of these details are not included in the outline.

Finally we are ready to write our algorithm in pseudocode. This consists of using keywords and indentation as needed.

multiplier = 0
Print "Enter a number: "
Get number
Print "The multiplication table for ", number, "is:"
Repeat 13 Times
  result = number * multiplier
  Print number, " x ", multiplier, " = ", result
  multiplier = multiplier +1
End Repeat
Print "Goodbye"

Step 4: Test the algorithm

Let’s test it with the following input:
5

<table>
<thead>
<tr>
<th>multiplier = 0</th>
<th>multiplier</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Print &quot;Enter a number: &quot;</td>
<td>Enter a number: 5</td>
</tr>
<tr>
<td>Get number</td>
<td>multiplier number</td>
</tr>
<tr>
<td>0 5</td>
<td></td>
</tr>
<tr>
<td>Print &quot;The multiplication table for &quot;, number, &quot;is: &quot;</td>
<td>The multiplication table for 5 is:</td>
</tr>
<tr>
<td>Repeat 13 Times</td>
<td></td>
</tr>
<tr>
<td>result = number * multiplier</td>
<td>multiplier number result</td>
</tr>
<tr>
<td>0 5 0</td>
<td></td>
</tr>
<tr>
<td>Print number, &quot; x &quot;, multiplier, &quot; = &quot;, result</td>
<td>5 x 0 = 0</td>
</tr>
<tr>
<td>multiplier = multiplier +1</td>
<td>multiplier number result</td>
</tr>
<tr>
<td>result = number * multiplier</td>
<td>1</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---</td>
</tr>
<tr>
<td>multiplier = multiplier + 1</td>
<td></td>
</tr>
<tr>
<td>Print number, &quot; x &quot;, multiplier, &quot; = &quot;, result</td>
<td>5 x 1 = 5</td>
</tr>
<tr>
<td>multiplier = multiplier + 1</td>
<td></td>
</tr>
<tr>
<td>Print number, &quot; x &quot;, multiplier, &quot; = &quot;, result</td>
<td>5 x 2 = 10</td>
</tr>
<tr>
<td>multiplier = multiplier + 1</td>
<td></td>
</tr>
<tr>
<td>Print number, &quot; x &quot;, multiplier, &quot; = &quot;, result</td>
<td>5 x 3 = 15</td>
</tr>
<tr>
<td>multiplier = multiplier + 1</td>
<td></td>
</tr>
<tr>
<td>Print number, &quot; x &quot;, multiplier, &quot; = &quot;, result</td>
<td>5 x 4 = 20</td>
</tr>
<tr>
<td>multiplier = multiplier + 1</td>
<td></td>
</tr>
<tr>
<td>Print number, &quot; x &quot;, multiplier, &quot; = &quot;, result</td>
<td>5 x 5 = 25</td>
</tr>
<tr>
<td>multiplier = multiplier + 1</td>
<td></td>
</tr>
<tr>
<td>Print number, &quot; x &quot;, multiplier, &quot; = &quot;, result</td>
<td>5 x 6 = 30</td>
</tr>
<tr>
<td>multiplier = multiplier + 1</td>
<td></td>
</tr>
<tr>
<td>Print number, &quot; x &quot;, multiplier, &quot; = &quot;, result</td>
<td>5 x 7 = 35</td>
</tr>
<tr>
<td>multiplier = multiplier + 1</td>
<td></td>
</tr>
<tr>
<td>Print number, &quot; x &quot;, multiplier, &quot; = &quot;, result</td>
<td>5 x 8 = 40</td>
</tr>
<tr>
<td>multiplier = multiplier + 1</td>
<td></td>
</tr>
<tr>
<td>Print number, &quot; x &quot;, multiplier, &quot; = &quot;, result</td>
<td>5 x 9 = 45</td>
</tr>
<tr>
<td>Print number, &quot; x &quot;, multiplier, &quot;, = &quot;, result</td>
<td>5 x 9 = 45</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>multiplier = multiplier +1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>multiplier</th>
<th>number</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5</td>
<td>45</td>
</tr>
</tbody>
</table>

| result = number * multiplier | |

<table>
<thead>
<tr>
<th>multiplier</th>
<th>number</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Print number, &quot; x &quot;, multiplier, &quot;, = &quot;, result</th>
<th>5 x 10 = 50</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiplier = multiplier +1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>multiplier</th>
<th>number</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>5</td>
<td>50</td>
</tr>
</tbody>
</table>

| result = number * multiplier | |

<table>
<thead>
<tr>
<th>multiplier</th>
<th>number</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>5</td>
<td>55</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Print number, &quot; x &quot;, multiplier, &quot;, = &quot;, result</th>
<th>5 x 11 = 55</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiplier = multiplier +1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>multiplier</th>
<th>number</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>5</td>
<td>55</td>
</tr>
</tbody>
</table>

| result = number * multiplier | |

<table>
<thead>
<tr>
<th>multiplier</th>
<th>number</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>5</td>
<td>60</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Print number, &quot; x &quot;, multiplier, &quot;, = &quot;, result</th>
<th>5 x 12 = 60</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiplier = multiplier +1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>multiplier</th>
<th>number</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>5</td>
<td>60</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>We have done the loop 13 times so go on</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Print &quot;Goodbye&quot;</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Goodbye</th>
</tr>
</thead>
</table>

### 3. THE CONDITION-CONTROLLED LOOP

The second type of repetition structure is the condition-controlled loop. This loop repeats as long as a given condition is true. It uses the keywords While and End While. There may be many statements within the body of the loop.

```plaintext
Print "Enter a number: 
Get number
While number < 0
    Print number, " is negative, please enter another number: 
    Get number
End While
Print "The loop is done."
```

We cannot tell how many times this loop will execute. It depends on the values that are entered by the user. The first two statements each execute once. Then processing moves on to the While statement. The first thing that happens in a While loop is a condition is tested. If the condition is true, processing moves to the statements inside the loop. If the condition is false, processing moves to the statement immediately following End While. Each time processing gets to the End While, it goes back to the While and tests the condition.
It is possible that a While loop will not execute. For example consider the following input:

<table>
<thead>
<tr>
<th>Print &quot;Enter a number: &quot;</th>
<th>Enter a number: 18</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get number</td>
<td>number 18</td>
</tr>
<tr>
<td>While number &lt; 0</td>
<td>the condition is false; go to statement after the loop</td>
</tr>
<tr>
<td>Print &quot;The loop is done.&quot;</td>
<td>The loop is done</td>
</tr>
</tbody>
</table>

Notice the loop was never executed. The condition was tested and found to be false so processing continued at the statement following the End While.

Now consider the input:

-7
-52
8

<table>
<thead>
<tr>
<th>Print &quot;Enter a number: &quot;</th>
<th>Enter a number: -7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get number</td>
<td>number -7</td>
</tr>
<tr>
<td>While number &lt; 0</td>
<td>the condition is true; do the loop</td>
</tr>
<tr>
<td>Print number, &quot; is negative, please enter another number: &quot;</td>
<td>-7 is negative, please enter another number: -52</td>
</tr>
<tr>
<td>Get number</td>
<td>number -52</td>
</tr>
<tr>
<td>While number &lt; 0</td>
<td>the condition is true; do the loop</td>
</tr>
<tr>
<td>Print number, &quot; is negative, please enter another number: &quot;</td>
<td>-7 is negative, please enter another number: -8</td>
</tr>
<tr>
<td>Get number</td>
<td>number 8</td>
</tr>
<tr>
<td>While number &lt; 0</td>
<td>the condition is false; go to statement after the loop</td>
</tr>
<tr>
<td>Print &quot;The loop is done.&quot;</td>
<td>The loop is done</td>
</tr>
</tbody>
</table>

Note the loop condition, (number < 0); it tests the value of a variable. That variable is the *loop control variable*. In the case of complex conditions, there can be more than one loop control variable, for example, (hours < 40 and status == "salaried").

When constructing a loop, three things must happen to the loop control variable. First, we must identify the variable(s) whose values need to be tested, that is, the loop control variable(s). Then we must construct a correct condition involving the loop control variable(s).

Next, the loop must be *primed*, meaning each loop control variable must have a starting value. Each loop control variable must be assigned a value before the loop begins. In our example, *number* is the loop control variable and a value is read into *number* before the loop begins.
Recall that variables can have "leftover" values and if we did not assure that \textbf{number} was assigned a value before the While, we could not be sure what was being tested and the results would be uncertain.

Lastly, it is critical that the value of the loop control variable change somewhere inside the loop. If the condition doesn't change, there is no way to stop (\textit{terminate}) the loop once it has begun. This produces a loop that repeats over and over. This is called an \textit{infinite loop}. The Get statement inside the loop in our example above allows the value of the loop control variable to change, making it possible for the loop to end.

4. SAMPLE ALGORITHM USING REPETITION STRUCTURES

Let's solve a problem that uses a condition-controlled loop.

Write an algorithm that gathers the annual sales from all the branches in the "First Chance Sales" company. The algorithm will ask the user for the annual sales for each branch. When a –1 is entered the algorithm will print a report that displays the number of branches reported and the total sales for the entire company.

The problem statement indicates that the user will enter a –1 to indicate that they are done entering branch information. When a specific value is used to stop a loop, it is called a \textit{sentinel value}. When using a sentinel value, it is important to select a value that would not be a normal value. It is not likely that a branch would have annual sales of –1, so the value will work for a sentinel value.

\textbf{Step 1 - Analyze the problem}

<table>
<thead>
<tr>
<th>input</th>
<th>given values</th>
<th>processing</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>annual sales for each branch</td>
<td>sentinel value of -1</td>
<td>count branches</td>
<td>number of branches</td>
</tr>
<tr>
<td></td>
<td></td>
<td>accumulate total sales</td>
<td>total sales</td>
</tr>
</tbody>
</table>

\textbf{Step 2: Outline the solution.}

After analyzing the problem, we see that we will be getting branch sales over and over. This indicates a repetition structure. Since we do not know how many times we will repeat this loop, we will use a condition-controlled loop.

When determining order we will need to remember to prime the loop and to change the value of the loop variable inside the loop. Also, determine which instructions are to go inside the loop and which are to be outside the loop.

\begin{itemize}
  \item initialize accumulator and counter
  \item get first branch sales amount
  \item loop as long as sales amount is not equal to –1
    \begin{itemize}
      \item increment the counter
      \item add this branch sales to total
    \end{itemize}
  \item get next branch sales amount
  \item end the loop
  \item print the count of branches
  \item print the total sales
\end{itemize}
Step 3: **Develop an algorithm**

Again we add the details, selecting meaningful variable names, adding input prompts and output labels. We specify the loop control variable and write a condition that is true except when the user enters a \(-1\). We determine appropriate initial values for the accumulator and the counter and write calculations for the accumulator and the counter.

\[
\begin{align*}
\text{total} &= 0 \\
\text{count} &= 0 \\
\text{Print } & \text{"Please enter the annual sales for the first branch "} \\
\text{Get branchSales} \\
\text{While (branchSales } \neq \text{ -1) } \\
\quad \text{count} &= \text{ count } + 1 \\
\quad \text{total} &= \text{ total} + \text{ branchSales} \\
\quad \text{Print } & \text{"Please enter the annual sales for another branch "} \\
\text{Get branchSales} \\
\text{End While} \\
\text{Print } & \text{"There were ", count, "branches that reported. "} \\
\text{Print } & \text{"The total sales for all branches was ", total}
\end{align*}
\]

Step 4: **Test the algorithm**

Let’s use the following input:

<table>
<thead>
<tr>
<th>3598723</th>
<th>total = 0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>984685</td>
<td>total</td>
<td>0</td>
</tr>
<tr>
<td>5893265</td>
<td>count = 0</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Print "Please enter the annual sales for a branch 
Get branchSales
While (branchSales != -1)
  count = count + 1
  total = total + branchSales
  Print "Please enter the annual sales for another branch 
Get branchSales
End While
Print "There were ", count , "branches that reported. 
Print "The total sales for all branches was ", total

<table>
<thead>
<tr>
<th>total = 0</th>
<th>total</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Please enter the annual sales for a branch \text{3598723}

Get branchSales

<table>
<thead>
<tr>
<th>total</th>
<th>count</th>
<th>branchSales</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>3598723</td>
</tr>
</tbody>
</table>

While (branchSales != -1)
  the condition is true; do the loop
  count = count + 1
  total = total + branchSales
  Print "Please enter the annual sales for another branch"

<table>
<thead>
<tr>
<th>total</th>
<th>count</th>
<th>branchSales</th>
</tr>
</thead>
<tbody>
<tr>
<td>3598723</td>
<td>1</td>
<td>3598723</td>
</tr>
<tr>
<td>Get branchSales</td>
<td>total</td>
<td>count</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------</td>
<td>-------</td>
</tr>
<tr>
<td>3598723</td>
<td>1</td>
<td>984685</td>
</tr>
</tbody>
</table>

While (branchSales != -1)
the condition is true; do the loop

<table>
<thead>
<tr>
<th>count = count + 1</th>
<th>total</th>
<th>count</th>
<th>branchSales</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>984685</td>
<td></td>
</tr>
</tbody>
</table>

total = total + branchSales
Print "Please enter the annual sales for another branch"
Please enter the annual sales for another branch
5893265

Get branchSales
Print "Please enter the annual sales for another branch"
Please enter the annual sales for another branch
5893265

While (branchSales != -1)
the condition is true; do the loop

<table>
<thead>
<tr>
<th>count = count + 1</th>
<th>total</th>
<th>count</th>
<th>branchSales</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>5893265</td>
<td></td>
</tr>
</tbody>
</table>

total = total + branchSales
Print "Please enter the annual sales for another branch"
Please enter the annual sales for another branch
-1

Get branchSales
Print "There were ",count,"branches that reported."
There were 3 branches that reported.
Print "The total sales for all branches was ", total
The total sales for all branches was 10476673

L. Multiple Control Structures in a Single Algorithm

We have already seen an example of an algorithm that uses both sequence and selection structures and another that uses both sequence and repetition. It is possible to use many various combinations of the three control structures in a single program. Making decisions about how these different structures are to be combined is an essential part of the design.

1. SERIAL COMBINATIONS

One way to combine a variety of structures is to put them one right after another as in the following example.

Write an algorithm that gets scores from the keyboard. Continue to get scores until a –1 is entered. Have the algorithm output the average of the scores entered. Be careful to protect against the possibility of dividing by zero. If no scores are entered, output a message that no scores were entered.
total = 0
count = 0
Print "Enter the first score: ", score
Get score
While (score != -1)
    total = total + score
    count = count + 1
    Print "Enter the next score: ", score
End While
If (count > 0)
    average = total / count
    Print "The average of these scores was ", average
Else
    Print "There were no scores entered."
End If

2. NESTED COMBINATIONS

It is also possible to put one structure inside another, for example, when a selection needs to be made inside a loop, or a loop is needed inside the true or false portion of a selection structure. This is called nesting. You can also nest an If statement inside the true or false portion of another If statement or a loop statement inside another loop.

When using nesting, it is important to maintain correct structure. To put a loop inside the true portion of an If statement, the entire loop, including the End While or End Repeat, must be inside. Similarly, when a loop must appear inside an Else, the entire loop, including the End While or End Repeat, must be inside the Else. Also, when a selection structure is inside another structure, the entire selection, including the Else (if present) and End If, must be inside the second structure.

An easy way to determine if your structure is correct is to draw a bracket connecting the beginning and end of each If, Else, and End If in a selection and a bracket connecting the beginning and end of each loop. The brackets should be nested, one inside another. If a bracket cuts through another bracket, your structure is incorrect. For example,

Consider the following example that uses nested structures:

Write an algorithm that reads from a file. The first number in the file is the number of scores there are in the file. Following this first number are the scores for Exam 1. The program is to read all the scores and report the number of students that passed the exam, the number of students that failed the exam, and the average exam score. A score of 60 or above is considered passing.
failCount = 0
passCount = 0
total = 0
Get numScores
Repeat numScores Times
    Get score
    If score < 60
        failCount = failCount +1
    Else
        passCount = passCount +1
    End If
    total = total +score
End Repeat
average = total / (failCount + passCount)
Print "There were ", passCount, " students that passed the exam."
Print "There were ", failCount, " students that failed the exam."
Print "The average score was ", average

So, there it is… your introduction to problem solving and the programming process. Keep up with your reading and the assignments, take an active part in class, see the tutors and your instructor for help when needed, and keep a positive attitude and you should find this a challenging, but satisfying experience. (When you come for help, bring along as much of your algorithm as you were able to produce.)

Good luck

© Linda DuHadway and Mary Veronica Kolesar, August 2003 (revised August 2006)