MC4060

Programming Fundamentals

Information Systems

Software Support & Training

Micron™

MC 4060 PROGRAMMING FUNDAMENTALS

Prerequisites

- MC6013 Windows NT 4.0 Getting Started or MC6012 Test-out for MC6013
- MC6015 Windows NT Explorer or MC6016 Test-out for MC6015

Course Hours & Sessions

Total Length: 5.5 hours

The course consists of two sessions: session one lasts 2.5 hours and session two lasts 3 hours.

Credit Requirements

- 100% attendance in both sessions, unless prior arrangements are made with the instructor(s).
- Completion of all in-class exercises and post-class assignments.

Test-Out

The test-out for MC4060 Programming Fundamentals will be available soon.

TABLE OF CONTENTS

PROGRAMMING FUNDAMENTALS - SESSION 1

Objectives

Goal

Establish a basic programming foundation for team members interested in advancing their programming knowledge and abilities.

Objectives for this Session

After completing this session, the student should be able to:

- Identify the basics of computing: input-process-output, common misconceptions, and user interfaces.
- Identify the major categories of programming.
- Recall four major methodologies of programming.
- Identify the basics of computer process flows: ordering, flow charts, and syntax.
- Identify and reconstruct basic programming techniques of input-output, variables, and commands.

INTRODUCTION

What is programming?

Computers must be told what to do. When you operate a computer, everything you use—within programs like Microsoft Word, or on systems like the VAX or Windows NT—has been previously programmed so the computer knows how to function. Programmers write "code" to instruct the computer step by step how to operate appropriately. Programming can be very simple, such as writing an application that looks up a worker on the MERC, or it can be very complex, such as writing the MICIS application that tracks and records all chemical data at Micron.

There is a misconception that programming is difficult and mysterious; programming is simply writing step-by-step procedures so an end user can transmit the appropriate information to a computer and vice versa. Obviously, computers are worthless unless you give them specific instructions that they can process input and output, as described below.

Input-Process-Output Model

The seemingly complex concepts of computers and programming are really not very complex at all when you recognize that these concepts are based upon the ordinary input-process-output model that you use in your daily life.

Examples of everyday input-process-outputs are:

	Input	**Process**	**Output**
Gas Station	Drive empty car in	Gas up, then pay	Drive full car out
Fast Food Restaurant	Drive in (or Eat in) hungry	Order food, pay for food, eat food	Drive away full
Discount Warehouse	Empty cart, full wallet	Find bulk items, pay for them, get them home	Full pantry and garage, empty wallet

In each of the examples, you take an input, add some value or service, and then become either fueled up, fed up, or stocked up.

Computer programming works in this fashion: you take raw data that is basically numbers (input), crunch data with programming (process), and then receive useful information (output).

One example of raw data might be a list of temperatures from the past year. With no explanation, this is just a bunch of random numbers. Put these numbers in a context: if you are planning an outdoor wedding for late October, you can review the temperatures recorded on October 27. Looking at temperatures from the past decade would aid you in making an informed processing decision. You may find Idaho is cold in late October, and you might want the wedding to be indoors. Now imagine a wedding planning program that could be taught this process and make these kinds of suggestions to you, the user, without forcing you to research the temperatures for October. The program could be expanded to look for the past 100 years' worth of temperatures on October 27 and the computer would be better able to predict the most likely temperature.

Following this input-process-output model, you will need to enter information into the computer. Some input devices that you might use include:

- Keyboard
- Mouse
- Optical Scanner
- Pen
- Touch Screen
- Bar Code Scanner
- Hard Drive (although not a real input device, you can get data from here)
- Network (a collection of computers together that can give data to your computer as input)

The computer program can then process this data. When you are done, you will want to get output from your computer. Some output devices that you might use include:

- Monitor
- Printer
- Hard Drive (although not a real output device, you can store data here)
- Network (a collection of computers together that can send data from your computer as output)

Common Misconceptions

One of the common misconceptions in computer programming is that computers make mistakes. We have to clarify this: mistakes do occur when people work with computers, but the computers do not innately make mistakes or we would not be able to sell them on the market.

"Mistakes" can be caused by:

- Poorly written computer programs: The programs that have been developed for the computer are incorrect and make many errors in the procedure or processing.
- Inaccurate data: We often use the phrase "Garbage In = Garbage Out" (GIGO). It is very important that you give computers the correct information in the beginning if you want to get the correct data out.
- Programming failure due to assumptions: What you intend the computer program to do and what it actually accomplishes might be two different things. The problem lies in the fact that people can sort out ambiguities or interpret basic knowledge and then convert them into logical sounding statements, but computers cannot.

User Interfaces

A user interface is the element of a computer program that can be seen (or heard or otherwise perceived) by the human user, along with the commands and processes that allow the user to control its operation and input data.You interact with user interfaces in your daily life when you use cell phones, pagers, copy machines, cars, and even microwaves. Each device has unique ways of conveying information to you, the user, and in return you interact with the device by inputting information with a keyboard or mouse. Computers are no exception. In early computing, user interfaces ranged in design and scope, but in recent years there has been a move to standardization. Character user interfaces (CUIs) and graphical user interfaces (GUIs) are two of the most common types of computers user interfaces.

A character user interface, commonly known as CUI (pronounced chewy), is used to describe character-based programming languages. CUI language environments were the only programming available in early computers. Users entered commands (instructions) into the computer, and then the computer executed those commands. This process was not hard to learn, but it was difficult to transfer from one computer system to another. CUIs are still essential to network-based programming and for some hard-core programmers. CUIs can be found at Micron by looking at VAX/VMS and DOS applications. While many programs that we use today were originally based upon CUI, most have evolved beyond that point.

A graphical user interface, commonly known as GUI (pronounced gooey), is used to describe modern computer interfaces, such as the Microsoft Windows operating system or common home applications like Quicken 2000. In these computer interfaces, you use both the mouse and the keyboard. The mouse allows you to select icons, buttons, and other objects by clicking on them. Keyboards are primarily used to input characters and numbers. To create order and organization within GUIs and make them usable, software manufacturers have developed standard icons and menus throughout their programs. You can learn the “look and feel” or the way the GUI looks only once, and then be able to apply it to every other program designed by that software manufacturer.

PROGRAMMING CATEGORIES

Before learning about specific programming languages, you should first understand the history of the various programming categories. A programming category is determined by the methods that the computer and the programming language use to interact with each other and by the methods you must use to program the computer. Each category has its advantages and disadvantages; the general behavior of each category sets one apart from another.

Binary Code & Machine Language

The chips that we manufacture at Micron are used to save information as input or output. The language of computers is really just the powering on and powering off of individual transistors that are etched onto the computer chips. Computers translate the "ons" into "1s" and the "offs" into "0s." For example, when the power is on for a light bulb, it lights up, and this is considered a 1. When the power is off, the bulb is not lit, and this is a 0. This concept of communicating with computers led to binary code, the language that you and the computer can both understand. Binary code is just the collection of the individual bits of 1s and 0s used to represent letters and eventually words. An example of a letter in binary code would be:

01100001 = a

01000001 = A

The above example illustrates a byte, which is 8 bits strung together to make one letter. Computers convert every instruction into these basic building blocks.

Now that you understand binary code and machine language, it is possible to program your computer manually by inputting all the 1s and 0s that represent letters and words. This would be a very difficult and daunting task: there is a high probability for error, it would take an extremely long time to complete, and when completed, only the computer could understand it. This machine language is so impractical that computer programmers invented several better methods to program computers.

Assembly

The creation of Assembly Language was one of the first attempts to simplify the computer programming process. The assembly language method incorporates some basic and understandable word combinations, such as MOV for move or CPY for copy, which can be easily read by others and are less prone to errors than machine language. The assembly method talks directly to the parts of the computer and requires an assembler to convert the simple commands into machine language. However, assemblers only function on the specific computer for which they are built, and writing a very sophisticated program requires many of the same commands to be repeated over and over again. Although assembly programs are more understandable than machine language, they fail to be efficient.

Portable Assembly

Portable assembly is very similar to assembly, but compilers replace the assemblers, allowing multiple computers to share the program. The C programming language was the first language to accomplish this task. Portable assembly is important because it can directly manipulate the computer hardware like machine code or assembly code.

The main disadvantage to the portable assembly method is that the programming code is not necessarily related to the functions; the C language was written by programmers for programmers, and it contains many commands that are hard to follow.

High-level Languages

High-level languages use more English language-related code statements that are easy to recognize and understand. For example, the command to print a document in a high-level language is often “Print” or some variation of that command. Programs written in these languages were very popular in early programming because they could be written quickly and were capable of performing complex tasks. However, programs created in high-level languages can quickly become very large, with a complex set of slow machine code instructions. They also cannot directly access the hardware like C, assembly, and machine code.

PROGRAMMING METHODOLOGIES

High-level programming languages are the most common type of programming used today. At Micron we use the high-level languages for the majority of our applications. These high-level languages can be broken down further into the three major programming methodologies: procedural programming, object-oriented programming, and markup programming.

Procedural Programming

Procedural programming languages are code intensive, requiring the programmer to specify step by step how the computer must accomplish a task. Procedural languages generally have strict rules of use and design because they must be precisely formatted so the computer can understand the appropriate functions. They generally require manual keyboard input for all code, similar to entering commands at a DOS or Unix prompt. Traditionally, programmers could only design CUI interfaces with procedural languages, but today these languages can produce a wide variety of both CUIs and GUIs. Examples of procedural programming languages include C, C++, Cobol, Fortran, and PERL. These are covered in detail in “Programming at Micron” on page 63.

Object-Oriented Programming (OOP)

The official definition of object-oriented programming (OOP) is a type of programming that encapsulates, or captures, a small amount of data along with instructions about how to manipulate that data. (Potter, 562) Object-oriented applications often have GUIs and this type of language is currently the fastest growing segment of programming, other than the Internet. Object-oriented programming (OOP) is a type of programming that works with “objects” rather than with the actions needed to accomplish a particular task, like in procedural languages. An object can be defined as an item that has properties to describe it. For example, you are an object. You contain properties that tell us more about you; you have a street address where you live, a Micron employee identity, certain skills, and relationships with people around you. OOP operates uniquely with these special object properties. First, you teach the computer all about a particular object. When the object is used in the program, the computer program already knows the object’s role and what it can do for us, just like when you are asked to accomplish a task for Micron. Examples of object-oriented programming include JAVA, Visual Basic, and Visual C++. These are covered in detail in “Programming at Micron” on page 63.

Markup Programming

Markup programming allows you to write plain text and enhance it (or mark it up) for use on another computer via the Internet. Hypertext Markup Language (HTML) is the first of such languages, and it has revolutionized the content of the Internet. Modern versions of markup languages are defined by worldwide organizations and are agreed upon as industry standards to ensure compatibility among a wide range of computers and browser programs. Examples of markup programming include HTML and Extensible Markup Language (XML). These are covered in detail “Programming at Micron” on page 63.

Database Programming

A database is a collection of information for later retrieval, such as your address book. Address books are organized by last name, and you can find a person's name, address, and phone numbers by searching for the last name. In the same way, databases are related pieces of information collected into individual records. To program a database, you need a language that is able to work with the unique database format. Some databases use their own internal languages such as SAP's ABAP language; others rely on universal languages such as SQL (Standard Query Language) that can exchange data between different systems.

Before you can program, you first have to think like a computer. That isn't as hard as it sounds. You might relate computer programming to teaching someone how to bake chocolate chip cookies. You have to be careful to use words that novice cooks can understand so they can follow the specific steps. You cannot assume understanding or skip steps because beginning chefs (or the computer) will not be able to understand.

Now if you break down the process of baking chocolate chip cookies to a simple step-by-step procedure, it might look like this:

```
1: Collect the ingredients: flour, baking soda, salt,
   butter, sugar, brown sugar, vanilla extract, eggs,
   chocolate chips, and optional nuts
2: Measure and combine the ingredients
3: Put the globs of dough on the cookie sheet
4: Bake the cookies
5: Let cool
6: Eat
```

If that were all there were to baking cookies, then we would be done. As everyone knows, the process of creating great cookies is much more complex. Not only do you have to follow the recipe carefully, but you also have to time certain events precisely to make the cookies turn out perfectly. To help us organize our thinking, we will be using a modeling tool called a flow chart.

Flow Charts

One of the most effective methods of diagramming complex programming logic is to use a flow chart. Flow charts allow you to explain the logic of your thinking so you can translate it for the computer's use.

If you use this class as an example of the inputs-process-outputs model discussed in the Introduction, you can create the following flow chart:

- You came into this class as an input.
- You are processed in the sense that you have to learn or be taught something.
- You are now an output that has changed because you have (presumably) grown smarter.

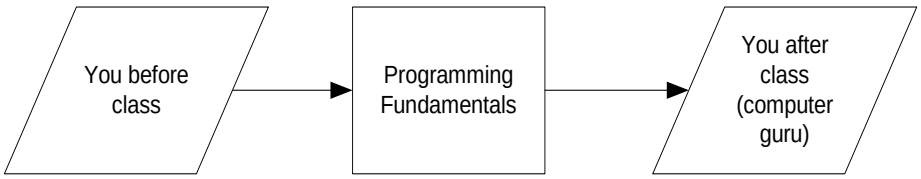
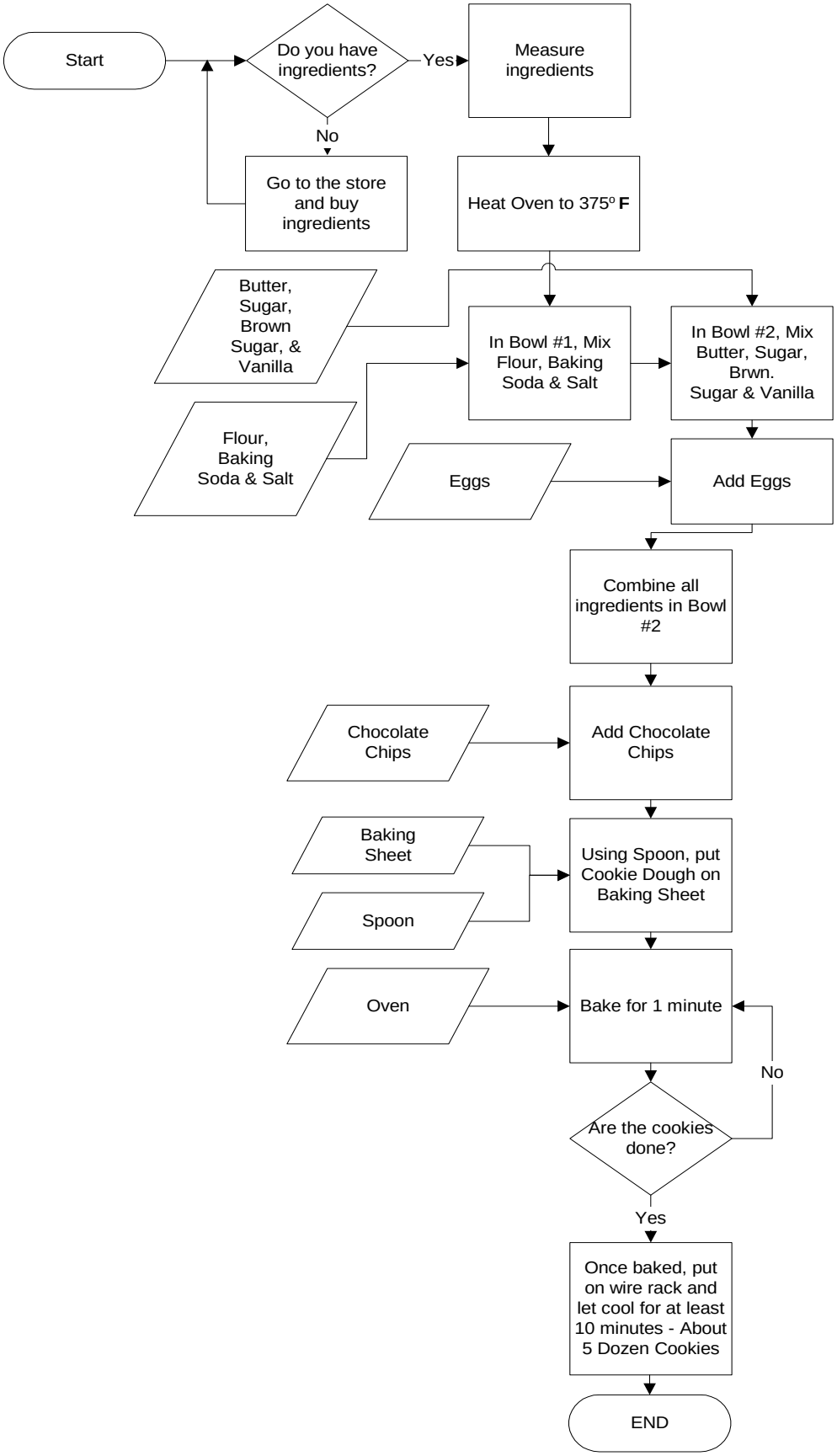


Figure 1: Diagram of Input-Process-Output

In our previous cookie example, we could have easily overlooked steps or erroneously assumed certain operations in the cookie-baking process. By creating a flow chart to break down the process, you can model the exact steps and create a checklist to ensure that you stay on the correct path.

Here is the cookie example adapted to the flow chart modeling tool:



There are four traditional flow chart symbols that are universally identified. They are the basic building blocks for any flow chart. The following information explains each symbol and its purpose.

Process

A process (represented by a rectangle) shows the major steps or stages. This symbol usually represents the actions in the process. In the cookie example, you notice that all the process symbols contain actions.

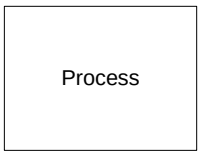


Figure 2: Process Rectangle

Decision

The decision tree (represented by a diamond with arrows) has as many as three choices: yes, no, or maybe. When programming a computer, avoid maybes because you cannot teach the computer to deal with ambiguities. Decision trees are the only objects in flow charting that are allowed more than one exit point.

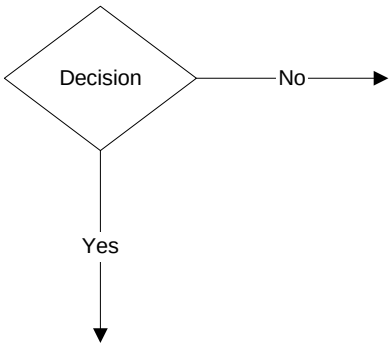


Figure 3: Decision Diamond

Data (Input/Output)

Data areas (represented by a rhombus) are usually inputs or outputs to the process boxes.

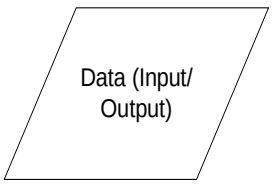


Figure 4: Data (Input/Output) Rhombus

Terminator

Terminators or End Points (represented by an oval) are the first or final stages of the process. If the end point for one process is an input for the next process, the terminator is a circle rather than an oval.

Figure 5: Terminator Ovals

When creating a flow chart, always follow these simple rules:

1. Always try to use standard flow charting symbols. We have described the four basic symbols (the rectangle, diamond, rhombus, and oval) in this manual. If your flow charts will be read by someone inheriting your project, try to follow the universal conventions, such as those found in Visio Technical, a flow charting program used here at Micron.
2. The logic in a flow chart should flow from top to bottom and from left to right.
3. Only the decision trees (diamonds) should have more than one exit point.
4. Decisions should always ask a Yes/No question. Computer logic does not understand anything other than these two states.
5. Use simple language in your flow chart. Avoid programming terms or lingo because they make the flow chart difficult to read. Sometimes just talking through or writing down issues will help you discover errors of logic or items that you assumed were covered.

Here is a collection of other flow chart items. Most any shape can be used, but if you create your own, be sure to include a key to the shape types, much like the following example:

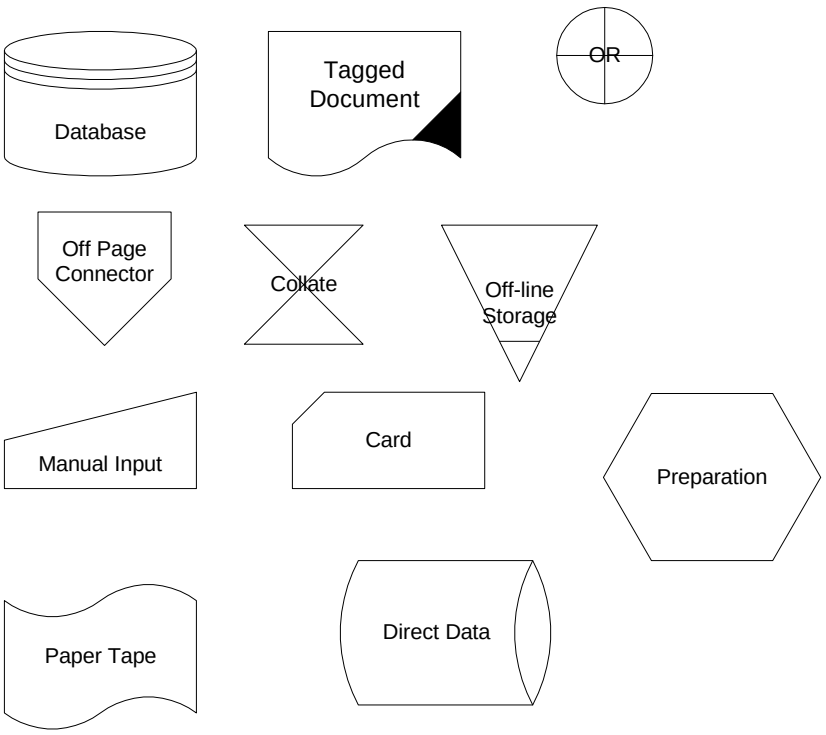


Figure 6: Key for Flow Chart Symbols

Made Up Programming Language - Phake

After creating your flow chart, you need to decide what programming language you will use. Remember that computers cannot process flow charts or English sentences.

To better understand programming, you need to try some programming techniques for yourself. This is an exercise using the "Phake" language which was developed for this class. Phake is a pseudo-code language that simulates real programming.

Phake is made up of three pieces:

- Actions (Commands) - items that do
- Things (Variables) - items that have something done to them
- Modifiers (Switches) - items that change the actions or reverse their direction

These key words are needed to use the Phake language to teach a computer how to bake cookies.

Actions (Commands)	**Things (Variables)**	**Modifiers (Switches)**
IF	pantry	No / Yes
GOTO	ingredients	For
PURCHASE	store	On / Off
MEASURE	oven	In / Out
CHECK	flour	To
TURN	baking soda	With
GET	salt	375 degrees F
OPEN [CLOSE]	bowl #1	10 minutes
BAKE	bowl #2	
EAT	refrigerator	
TAKE	sink	
END	butter	
PUT	sugar	
ADD	brown sugar	
SET	vanilla	
MIX	eggs	
WAIT	chocolate chips	
	cookie dough	
	baking sheet	
	spoon	
	cooling rack	
	temperature	
	cookies	

The Phake language has the following rules (syntax):

1. Every line must start with an action (command). These actions can have modifiers (switches) that change their direction, but the action itself is essentially the same.
2. Every line must have an item that is acted upon; we will call these "things" (variables).

3. You are limited to the items on the list, and you cannot make up new actions or generate new things.
4. You can use linking words, such as **and** / **or**.

Using the Phake language, create a program to describe the process of baking cookies. Below is a typical example of what you may come up with. Do not worry if yours looks different; there are many solutions to the same problem in programming.

In the following program, the Phake code includes step numbers to the left to aid in line item differentiation.

Example 1: Phake language used to describe baking cookies

```
1:  CHECK pantry For flour, baking soda, salt, sugar,
    brown sugar, vanilla, chocolate chips
2:  CHECK refrigerator For eggs, butter
3:  SET ingredients = flour, baking soda, salt, butter,
    sugar, brown sugar, vanilla, chocolate chips, eggs
4:  IF No ingredients, GOTO store and PURCHASE
    ingredients
5:  GET ingredients Out pantry
6:  MEASURE ingredients
7:  TURN On oven and SET temperature 375 degrees F
8:  GET bowl #1
9:  GET spoon
10: MIX flour, baking soda, salt In bowl #1 With spoon
11: GET bowl #2
12: MIX butter, sugar, brown sugar, vanilla In bowl #2
    With spoon
13: ADD eggs To bowl #2
14: MIX ingredients In bowl #2 With spoon
15: ADD ingredients In bowl #1 To ingredients In bowl
    #2 With spoon
16: MIX ingredients In bowl #2 With spoon
17: ADD chocolate chips To bowl #2
18: MIX bowl #2 With spoon
19: GET spoon, baking sheet
20: PUT cookie dough On baking sheet With spoon
21: OPEN oven
22: PUT baking sheet With cookie dough In oven
23: CLOSE oven
24: BAKE 10 minutes
25: OPEN oven
26: GET Out cookies and cookie sheet
27: PUT cookies On cooling rack
28: PUT cookie sheet In sink
29: CLOSE oven
30: TURN Off oven
31: WAIT 10 minutes
```

```
32: EAT cookies
33: END
```

In this second example, you define all of the “things” (variables) used in the program at the beginning of the program. This process teaches the terms to a new cook; in the previous example you assumed that the cook knew the terms for each ingredient. Also, when you create variables, you can shorten their names to save on space, which was very important in early computing.

Example 2: Phake language used to describe baking cookies

```
1:  SET FL = flour, BS = baking soda, S = salt
2:  SET B = butter, Su = sugar, BSu = brown sugar
3:  SET V = vanilla, CC = chocolate chips, E = eggs
4:  CHECK pantry For FL, BS, S, Su, BSu, V, CC
5:  CHECK refrigerator For E, B
6:  SET ingredients = FL, BS, S, B, Su, BSu, V, CC, E
7:  IF No ingredients, GOTO store and PURCHASE
    ingredients
8:  GET ingredients Out pantry
9:  MEASURE ingredients
10: TURN On oven and SET temperature 375 degrees F
11: GET bowl #1
12: GET spoon
13: MIX FL, BS, S In bowl #1 With spoon
14: GET bowl #2
15: MIX B, Su, BSu and V In bowl #2 With spoon
16: ADD E To bowl #2
17: MIX ingredients In bowl #2 With spoon
18: ADD ingredients In bowl #1 To ingredients In bowl
    #2
19: MIX ingredients In bowl #2 With spoon
20: ADD CC To bowl #2
21: MIX bowl #2 With spoon
22: GET spoon, baking sheet
23: PUT cookie dough On baking sheet With spoon
24: OPEN oven
25: PUT baking sheet With cookie dough In oven
26: CLOSE oven
27: BAKE 10 minutes
28: OPEN oven
29: GET cookies and cookie sheet Out
30: PUT cookies On cooling rack
31: PUT cookie sheet In sink
32: CLOSE oven
33: TURN Off oven
34: WAIT 10 minutes
35: EAT cookies
```

```
36: END
```

Post-Phake Language

Variables

In our Phake language, we had a category called things that represented the items that the program needed to use. Another way to describe those things is to use the term “variable.” A variable is a memory location referred to by a name, but you could think of it as a container that holds information. In keeping with our cookie baking example, a typical variable might be like a measuring cup that can hold the proper amount of milk until you need it. The other advantage to a variable is that it can change; when you want your measuring cup to contain something else, it can be emptied and washed. Now it can hold flour.

In programming, a variable might be a number, dollar amount, or word that can change throughout a program, depending on how you use variables. Different programming languages will use the rules of the language to create whatever kind of variable is needed. The value of a variable can change during execution of a program by input and output. (Bradley, 660, 687-689)

Constants

Constants hold a specific piece of information throughout the program. In our cookie baking example, the sugar bowl is a constant. Unlike the measuring cup, a sugar bowl only holds one item (sugar), and it will always hold the same contents.

Literal

Literals are exact, do not change, and can be either numbers or characters. In the Phake language, when you set the variable FL = flour, FL was the variable and flour (the value contained within the variable) was the literal.

Commands

In our Phake language, we called the main drivers of the program actions. In most programming languages, these actions are called commands or key words. You will need to know how to modify or alter the commands to fit your needs. An example of a typical command might look like this:

LASERPRINT “my document”

This command prints your document to the laser printer of your choice.

Switches or Options

Switches or Options can be added to commands. In the Phake programming language, we called them modifiers. In most programming languages, these modifications are based upon a set of rules (syntax). We can take the previous example and add a switch to tell the laser printer to print landscape:

LASERPRINT - L “my document to print landscape”

The online help for your programming language lists the correct usage of switches or options for your commands.

Syntax

The term syntax refers to the rules or sequence that a programming language must follow when the code is written. When speaking your native language, you don't think about syntax because you instinctively know the rules. But when you learn a foreign language, you may need to learn new syntax rules. For example, in English the adjective comes before the noun—you would say "the red car," but in Spanish, the adjective comes after the noun, so you would say, "el auto rojo" which literally translates to "the car red." English speakers who are learning Spanish need to remember to put the adjective after the noun. If the word order is wrong, the meaning might not be clear. The same concept of syntax is used in computer programming languages.

In our Phake language, syntax was defined in rule #1 (refer to the Rules on page 17). Rule #1 stated that every line must start with a command (action). This is a simple example of a syntax rule. Rules in programming languages range from simple to very complex and exact. Using proper programming syntax is just as important as learning the terms themselves.

Now that you are thinking like a computer, the next challenge is to work with an actual programming language. One of the first languages that most programmers learn is a language called BASIC. In the early 1960s, BASIC ran only on main frame systems, but was redesigned in the late 1970s to give everyone the same exposure to programming on personal computers. BASIC has evolved from very rudimentary beginnings.

There are many companies that have developed BASIC programming languages, and they range from free products to licensed ones. QBasic or Quick BASIC is the latest version of the BASIC programming language from Microsoft. It falls somewhere between a standard CUI and a GUI because it has some characteristics of each interface. Although QBasic is not used as an official language here at Micron, it will aid you in identifying and applying programming fundamentals. Many of the commands used in QBasic are universal to all languages, or there are equivalents that accomplish the exact same goals. You should start with simple programs that accomplish quick tasks and then slowly build on your previous experience by trying more complex programming.

Running QBasic

For this class you will be running QBasic from a classroom network drive.

1. Click **START** > **RUN...**
2. In the Open field, type **F:\Basic\qbasic**
3. Click **OK.**
4. Press **<ESC>** to clear the entry dialog box.

After completing the instructions above, the QBasic welcome screen displays.

Figure 7: QBasic Welcome Screen

If you need further instructions on running QBasic on various Microsoft platforms, refer to “Appendix 1 - Additional QBasic Resources” on page 66.

Programming Input - Output

Your first simple QBasic program will use the input-process-output model that you learned earlier. In this program, you will create a command that accepts input. This should be easy because the command in QBasic is INPUT, and the command in QBasic to send information out to the monitor is PRINT.

New Command and Syntax Summary:

Command	Definition
PRINT	Sends information to the screen
INPUT	Receives input from the keyboard and assigns it to a variable
END	Finishes the program

Syntax	Definition
“ “	Quotes separate the string information from the code
;	Separates the information being printed from the variable
$	Designates the variable *name* as a string variable (those that contain words)

Follow these instructions to create your first program:

1. Open QBasic.
2. Type the following code exactly as written. Be sure to press the **<ENTER>** key after each line.

```
PRINT "What is your name?"
INPUT name$
PRINT "Hi "; name$
END
```

After typing the code, press the **<SHIFT>** and **<F5>** keys simultaneously to run the program. You are now an official computer programmer!

Here is the step-by-step breakdown of your first program:

- **PRINT** Prints *What is your name?* on the screen.
- **INPUT** Accepts the information from the keyboard and assigns the information to the name$ variable.
- **PRINT** Prints Hi, [the name you entered].

- **END** Stops the program.

Saving a Program in QBasic

If you are happy with the results of the program, it is a good idea to save your work. Save the previous program as **FIRST.BAS**. When naming a QBasic file, use the DOS-based standards of eight characters or less. When saving QBasic programs for class, please save them to your **F:\BASIC** directory.

To save a program in QBasic

Using the mouse:

1. Click **FILE**.
2. Click **SAVE**.
3. Enter the name **FIRST.BAS**.
4. Click **OK**.

Using the keyboard:

1. Press **<ALT>** + **<F>**.
2. Press **<S>** for Save.
3. Enter the name **FIRST.BAS**.
4. Click **OK**.

Loading a File in QBasic

After you have saved a program, it is important that you know how to retrieve it. In this case, you will retrieve a class example file named **NUMBERS.BAS**:

1. Click **FILE**.
2. Click **OPEN**.
3. Choose **NUMBERS.BAS** from the list, or type in **NUMBERS.BAS**.
4. Click **OK**.

Note: For this class, the programs you will work with are supplied in the **F:\BASIC** directory. You have the option of either typing the examples for yourself or retrieving these files. For the remaining programs, the code will include step numbers on the left to aid in line-item differentiation.

Variables as Numbers

Computers can deal with numbers, but they need to be taught what to do with the numbers. For example, when you teach a child to add, you might say: “Let’s remove 1 apple from basket #1 and add it to the 3 apples in basket #2. Then we will have 4 apples in basket #3.” Like children, computers can evaluate these equations; computers just need to be taught how.

QBasic uses the following standard operators to deal with numerical variables:

- addition (+)
- subtraction (-)
- multiplication (*)
- division (/)

The mathematical formula for the apple example “Variables as Numbers” on page 24 looks like this:

1 + 3 = 4

With a syntax of:

a + b = c

Now we can apply variables from the example to the syntax:

Basket#1 + Basket#2 = Basket#3

The following is an example of how this might be written in a standard computer program. The line numbers are shown only to designate the separate steps.

```
1:  Basket#1 = 1
2:  Basket#2 = 3
3:  Basket#3 = Basket#1 + Basket#2
```

The syntax for the computer is:

```
1:  define a
2:  define b
3:  c = a + b
```

New Command and Syntax Summary:

Command	Definition
REM	Remarks or statements that are not included in the code, but are used for those observing the code later.
CLS	Clears the screen of all writing.

Syntax	Definition
:	Allows the programmer to put multiple commands on a single line of code. Be careful with its use because many commands compressed onto one line are difficult to follow or read later.

Example Program Name: **NUMBERS.BAS**

```
1:  REM This works with simple numbers
2:  CLS
3:  PRINT "Enter two numbers that you would like to
    multiply"
4:  PRINT
5:  PRINT "Enter your first number": INPUT x
6:  PRINT "Enter your second number": INPUT y
7:  z = x * y
8:  PRINT "The first number multiplied by the second
    number is"; z
9:  END
```

Program Notes:

- Line 1 is a command called REM, which is an abbreviation for remark; these are important remarks for the programmer to read later.
- Line 2 is a command called CLS, which clears the screen before you type anything.
- Line 3 is a print to the screen command.
- Line 4 is a simple PRINT statement to move the program down a line.
- In lines 5 and 6, the INPUT commands are put on the same line as each of the PRINT statements. This was included to show that multiple commands can occur on a single line, but it is only recommended when it makes the program clearer. In this case, the inputs are tied to the print statements so they are a natural fit.
- Line 7 sets the numeric variable ***z*** to be the product of taking variable ***x*** (your first number) multiplied by the variable ***y*** (your second number).
- Line 8 sends the output.
- Line 9 ends the program.

Exercise 1: Change the operator in **NUMBERS.BAS** to addition. Be sure to change the user instructions so they reflect the functional differences of the program.

In this program we used ***x***, ***y***, and ***z*** as the variable names. In early programming, using simple letters was the accepted practice because they were short and easy to find. Programming today has changed to use long descriptive variable names. For the example above, variable ***x*** could be ***firstuserinput*** and ***z*** could be ***multipliednumbers*** to give the programmer and any subsequent readers a hint as to what the function of the variable is.

TIP: Never use spaces in the names of your variables. Most programming languages do not understand the spaces. If you use spaces, your program may not be compatible with other programs or operating systems. In most programming languages if you do need separation for clarity, use the underscore character (_). However, QBasic does not support underscores. :-(

Advanced Variables as Numbers

You can use numerical values in complex ways. For example, consider the following formula for calculating body mass index (BMI). The United States National Institutes of Health says that body mass index (BMI) ratings can help you with your plans for weight loss and overall health. It is not easy to divide your weight in kilograms by your height in meters squared.

$$\frac{\text{Your Weight in Kilograms}}{\text{Your Height in Meters}^2} = \text{Body Mass Index}$$

The challenge for the programmer is to collect data in English units (measurements that are user friendly), convert it to metric measurements, and then perform the calculation. It is a good example of taking basic input data, processing it using a program, and then giving a beneficial output to the user.

New Command and Syntax Summary:

Command	Definition
USING	Used with PRINT to format the output of the variable to fit the format that is expected.
'	This little mark replaces the REM statement in later versions of BASIC.

Syntax	Definition
##.#	Creates a template for the USING command to display the information. This represents two digits, the decimal, then one last digit.

Example Program Name: **BMI.BAS**

```
1:  ' This works with complex numbers
2:  '
3:  '
4:  CLS
5:  PRINT "Enter your height in feet only"
6:  PRINT "For example, if you are 5 foot 7 inches,
    only enter 5": INPUT feet
7:  PRINT "Enter your remaining height in inches only"
8:  PRINT "From the example above, if you are 5 foot 7
    inches, only enter 7": INPUT inches
```

```
 9:  height1 = (feet * 12) + inches
10: PRINT "Enter your weight in pounds": INPUT weight1
11: weight2 = weight1 / 2.2
12: height2 = (height1 * 2.54) / 100
13: height3 = height2 * height2
14: ratio = weight2 / height3
15: PRINT "Your Body Mass Index ratio is: "; USING
    "##.#"; ratio
16: PRINT
17: PRINT "The Government Standards are below:"
18: PRINT " Ideal BMI is 20 to 24"
19: PRINT " 25 to 29 are advised they are possibly over
    their ideal weight"
20: PRINT " Those over 30 are advised to consult with
    their doctors"
21: PRINT " * NOTE: This is only one indicator of
    overall health *"
22: PRINT " * and should only be considered as a
    guideline.    *"
23: END
```

Program Notes:

- Lines 1 to 3 are comments for the programmer to read later.
- Line 4 clears the screen.
- Line 5 gives instructions.
- Line 6 has you input your height in feet only.
- Line 7 gives instructions.
- Line 8 has you input your remaining height in inches only.
- Line 9 sets the variable height1 to the product of your height in feet multiplied by 12 added to the number of inches that was specified. If you improved this program, you would make 12 a constant, so that if the number of inches in a foot changed, you could quickly change the program. If you scoff at the number of inches in a foot changing, remember that the Y2K bug was based upon the assumption that a constant date format would never change.
- Line 10 has you input your weight in pounds.
- Line 11 converts your weight from pounds to kilograms.
- Line 12 converts your height in inches to centimeters, then to meters.
- Line 13 squares your height (this could be used by a function or a power operator, but this is just a simple example).
- Line 14 actually calculates the BMI.
- Line 15 displays the final results to the screen in the format that you wanted.
- Lines 16 through 22 consist of a chart to aid you in understanding the output.
- Line 23 ends the program.

Variables as Characters (Strings)

Variables that contain anything from the keyboard, including letters, symbols, and even numbers, are often called strings. (Wang, 108) If a variable contains a word, it is string variable. Different types of variables are available to help the programmer and the computer keep track of what kind of data the variable contains.

Standard variables use standard math computations, but string variables (which can contain anything), have other rules to help the computer understand what to do. The rules for handling strings vary from one programming language to another. The following is an example of typical information stored in a string variable:

myvariablename$ = “this is the stuff that goes into a string variable”

Note the **$** character. This syntax shows the computer that myvariablename is a string. Most programming languages use this universal convention.

New Command and Syntax Summary:

Command	Definition
SPACE$(n)	Inserts the number of spaces that are specified in the parentheses ( ).

Syntax	Definition
+	When working with string variables, the + represents a new type of command, concatenate. In many other languages, there is special syntax for this operation.
$	Designates a string variable.

Example Program Name: **STRING1.BAS**

```
1:  ' This is a sample string variable handling program
2:  '
3:  '
4:  CLS
5:  PRINT "What is your first name": INPUT first$
6:  PRINT "What is your last name": INPUT last$
7:  name$ = first$ + SPACE$(1) + last$
8:  PRINT "Your name is "; name$
9:  END
```

Program Notes:

- Lines 1 to 3 are comments for the programmer to read later.
- Line 4 clears the screen.
- Line 5 has you input your first name.
- Line 6 has you input your last name.
- Line 7 combines your first name, a space, and your last name into a string variable called name$.
- Line 8 prints the contents of the string variable name$ to the screen.
- Line 9 ends the program.

Exercise 2: Modify **STRING1.BAS** so that the users also must enter their middle initial.

Advanced Variables as Characters (Strings)

For the next program, **STRING2.BAS**, the previous example is expanded to include more complex string handling routines.

New Command Summary:

Command	Definition
LEFT$(variable$, n)	Cuts the string variable$ by (n) the number of characters from the left.
UCASE$(variable$)	Changes the string variable$ to all uppercase letters.
LCASE$(variable$)	Changes the string variable$ to all lowercase letters.

Example Program Name: **STRING2.BAS**

```
1:  ' This is another string variable handling program
2:  '
3:  '
4:  CLS
5:  PRINT "What is your first name": INPUT first$
6:  PRINT "What is your middle name": INPUT middle$
7:  PRINT "What is your last name": INPUT last$
8:  name$ = first$ + SPACE$(1) + middle$ + SPACE$(1) +
    last$
9:  Upname$ = UCASE$(name$)
10: Lname$ = LCASE$(name$)
11: micron$ = LCASE$(LEFT$(first$, 2) + LEFT$(middle$,
    2) + LEFT$(last$, 10))
12: PRINT
13: PRINT "Your name is "; name$
14: PRINT
15: PRINT "Your name is "; Upname$; " in uppercase."
```

```
16: PRINT
17: PRINT "Your name is "; Lname$; " in lowercase."
18: PRINT
19: PRINT "Your Micron username should be "; micron$; "
    "
20: END
```

Program Notes:

- Lines 1 to 3 are comments for the programmer to read later.
- Line 4 clears the screen.
- Line 5 has you input your first name.
- Line 6 has you input your middle name.
- Line 7 has you input your last name.
- Line 8 combines your first name, a space, your middle name, a space, and your last name into a string variable called name$.
- Line 9 creates a new string variable called Upname$ that stores the upper case version of name$ (your name).
- Line 10 creates a new string variable called Lname$ that stores the lower case version of name$ (your name).
- Line 11 creates a new string variable called micron$ that takes the first letter from your first name, the first letter from your middle name, and the first six letters from your last name in an effort to guess your Micron username.
- Lines 13, 15, 17, and 19 print the results to the screen.
- Line 20 ends the program.

Exercise 3: Modify **STRING2.BAS** to correctly produce your Micron username.

Fun with String Variables

The next program you'll try is a Mad Lib program. As you become more advanced with BASIC programming, you may want to rewrite this code for practice. This example takes the variables defined in the first part of the program and rearranges them into a funny substitution sentence at the end of the program. Defining the variables at the start of the program is required by some programming languages. Even if it is not required, it is a very good practice to follow to give others visibility to the variables that will be used throughout the program.

New Command Summary:

Command	Definition
DIM	Creates and defines new variables for use in the program.

Example Program Name: **MADLIB1.BAS**

```
1:  '  This is my program called madlib1.bas
2:  '  I wrote it for my Programming Fundamentals Class
3:  '
4:  CLS
5:  DIM actioning$, animal$, someone$, funny$, farm$,
    bodypart$
6:  PRINT "Here is a little quiz.  Please use funny
    words,"
7:  PRINT "because they make this more interesting."
8:  PRINT
9:  PRINT
10: PRINT "Enter an action word ending in ING": INPUT
    actioning$
11: PRINT "Enter an exotic animal name": INPUT animal$
12: PRINT "Enter your friend's name": INPUT someone$
13: PRINT "Enter a funny sound": INPUT funny$
14: PRINT "Enter a farm animal": INPUT farm$
15: PRINT "Enter a body part": INPUT bodypart$
16: CLS
17: PRINT
18: PRINT
19: PRINT "One day many years ago, "; someone$; " was
    "; actioning$; " when suddenly there was a(an) ";
    funny$; " sound, much like a "; animal$; ".  ";
    "Frightened and scared "; someone$; " tripped over
    a "; farm$; "."; " In haste to get away, ";
    someone$; " broke his/her "; bodypart$; "."
20: END
```

Program Notes:

- Line 5 uses the new command DIM to establish the string variables. This programming practice is helpful to those who read this program later because they can find all of the variables that are used, without having to search through the entire program.
- Notice that the string variables are based upon the responses to the questions. These are by no means a guarantee of what the users will enter, but they help the programmer to write the little story in Line 19.

Variable Types Summary

The following table summarizes when to use the different variable types:

Variable Type	When to use
Numeric (Standard) Variables	Used when you want to treat the variable and its contents, the literal, as numerical information. Numeric variables can be added, subtracted, multiplied, divided, and any other function that you can perform on a number.
String Variables	Used when you want to treat the variable and its contents, the literal, as text information. String variables can be combined or compared, but they cannot have mathematical functions applied to them.

There are many other variables types and uses, but these are the major ones used in most programming languages. More information about variables specific to QBasic can be found in “Appendix 1 - Additional QBasic Resources” on page 66.

Logic Statements (Part One)

Often, you need some extra commands in your code to process the flow of the computer program. These are called the logic statements or the control commands. These commands can change the direction of the program or evaluate the input from the users. The follow statement is one possible logic command; you will find this command in most programming languages.

IF THEN Statement

An example of an IF THEN statement can be found on the flow chart for the cookie baking scenario. (See page 14.) In the flow chart, you determine whether you have certain ingredients. If you need them, then you are directed to buy them at the store. If you already have the ingredients, then the program continues to the measuring stage. In programming, this is called an IF THEN statement. Usually you are only testing for one condition at a time.

For example:

```
1: IF NO ingredients, THEN GOTO store and purchase
   ingredients
```

Or you could evaluate it like this:

```
1: IF YES ingredients, THEN GOTO measure ingredients
```

Notice that both of these statements are equivalent to each other. Each branch takes the information and routes you in the correct direction. Be careful with this kind of statement because it is easy to make a mistake in the logic of the IF THEN command. Review the following example:

```
1: IF YES ingredients, THEN GOTO store and purchase
   ingredients
```

This statement says, if you have the ingredients, then you will go to the store and purchase them; but even after you get them, you find that you have to go to the store over and over. Often, the logic flaw is not this easy to see, and it may take the programmer some time to catch the mistake.

This next example is similar to the one used in “Fun with String Variables” on page 31, but this time you will use IF THEN statements to discover the gender of your “Mad Libs victim.” This allows you to customize the final message.

New Command Summary:

Command	Definition
IF THEN	IF THEN works on the condition; if the statement is true, then the action that follows will occur.
GOTO	Takes you to a specific line or location.

Example Program Name: **MADLIB2.BAS** - with IF THEN Statements

```
1:  '  This is my program called madlib2.bas
2:  '  I wrote it for my Programming Fundamentals Class
3:  '
4:  CLS
5:  DIM actioning$, animal$, someone$, funny$, farm$,
    bodypart$, gender$, pronoun$
6:  PRINT "Here is a little quiz.  Please use funny
    words,"
7:  PRINT "because they make this more interesting."
8:  PRINT
9:  PRINT
10: PRINT "Enter an action word ending in ING": INPUT
    actioning$
11: PRINT "Enter an exotic animal name": INPUT animal$
12: PRINT "Enter your friend's name": INPUT someone$
13: PRINT "What gender is, "; someone$; "? Enter Male
    or Female": INPUT gender$
14:         IF gender$ = "Male" THEN pronoun$ = "his":
    GOTO Done
15:        IF gender$ = "Female" THEN pronoun$ = "her":
    GOTO Done
16:         IF gender$ = "" THEN pronoun$ = "his/her":
    GOTO Done
17:         PRINT "BAD ENTRY, ERROR.": pronoun$ = "his/
    her"
18: Done:
19: PRINT "Enter a funny sound": INPUT funny$
20: PRINT "Enter a farm animal": INPUT farm$
```

```
21: PRINT "Enter a body part": INPUT bodypart$
22: CLS
23: PRINT
24: PRINT
25: PRINT "One day many years ago, "; someone$; " was
    "; actioning$; " when suddenly there was a(an) ";
    funny$; " sound, much like a "; animal$; ".  ";
    "Frightened and scared "; someone$; " tripped over
    a "; farm$; ". "; " In haste to get away, ";
    someone$; " broke "; pronoun$; " "; bodypart$; "."
26: END
```

Program Notes:

- In **MADLIB2.BAS** you will find the GOTO statement and a line location (the mysterious statement “Done” on our reference line 18). It was used in this example to illustrate how to jump from one spot in the code to another. In later examples, the GOTO command has been replaced because it allows the programmer to jump from place to place without much regard to programmers who may inherit the application. It is recommended to only use the GOTO statement when absolutely necessary and, if used, to comment the location and the reasoning in your work.

IF THEN ELSE IF Statement

IF THEN usually has another option called ELSE IF. The IF THEN ELSE IF command allows you to have multiple IF THEN statements in a row to cover all of the possible combinations, instead of only working with yes/no type questions like the standard IF THEN.

In the following example, you are going to the store for a few things:

```
1:  IF I purchase the gummy bears THEN I have a snack
2:  ELSE IF I purchase the fruit snacks THEN I have a
    snack
3:  ELSE IF I purchase the chocolate covered peanuts
    THEN I have a snack
4:  ELSE I haven't bought a snack THEN I don't have a
    snack
5:  END IF
```

On the end of this example, notice the END IF. This returns the program back to where you left the original branch of logic or flow of the program.

The following demonstrates an example of IF THEN statements with a new command called ELSE IF.

New Command and Syntax Summary:

Command	Definition
ELSE IF	Allows you to continue with an IF THEN statement by giving you more choices.
ELSE	The final ELSE IF is designated with a single ELSE statement to signal the end of the logic.
END IF	Stops the IF THEN ELSE IF string of commands. This has replace the GOTO command.

Syntax	Definition
Return after each THEN statement	The format for IF THEN ELSE IF must be followed as it is shown below.

Example Program Name: **STEAK.BAS**

```
 1:  '  Cooking a Steak
 2:  '  IF THEN ELSE IF example
 3:  '
 4:  CLS
 5:  DIM cooklevel$, cooktime
 6:  PRINT "How do you want your steak cooked?"
 7:  PRINT "Please choose from rare, medium, medium
     well, and well done": INPUT cooklevel$
 8:          IF cooklevel$ = "rare" THEN
 9:          cooktime = 2
10:          ELSEIF cooklevel$ = "medium" THEN
11:          cooktime = 5
12:          ELSEIF cooklevel$ = "medium well" THEN
13:          cooktime = 9
14:          ELSEIF cooklevel$ = "well done" THEN
15:          cooktime = 15
16:          ELSE cooktime = 20
17:          END IF
18: PRINT "Cook the steak for"; cooktime; " minutes on
    each side for a perfect steak"
19: END
```

Program Notes:

- Lines 1 to 3 are comments.
- Line 4 clears the screen.
- Line 5 sets up the two variables to be used in the program.
- Line 6 sends the question to the user.
- Line 7 gives the options for the input.
- Lines 8 and 9 find that if the item is rare, then set the variable cooktime to 2. If not, it is to continue to evaluating lines.
- Lines 10 through 15 continue like lines 8 and 9.
- Line 16 sets the variable cooktime to 20 if the input does not match lines 8 through 15
- Line 17 ends the IF THEN loop and continues with the program.
- Line 18 sends the output to the user.
- Line 19 ends the program.

Exercise 4: Add a feature in **STEAK.BAS** so the program accepts cooklevel$ entries with any capitalization combination. For example, the program would accept RARE, Rare, or rare as correct entries. **Hint:** you only need to enter one line of code!

TIP: When working with IF THEN ELSE IF, try to plan for all possibilities. Notice in this example that only the four choices are listed. Good questions to ask yourself might include "What if the user entered nothing?" or "What if the entry is not even a word?"

Exiting QBasic

After you have saved your work, you can exit QBasic.

Using the mouse:

1. Click **FILE**.
2. Click **EXIT**.

Using the keyboard:

1. Press **<ALT>** + **<F>**.
2. Press **<X>** for Exit.

HOMEWORK 1

To successfully complete this course, this homework assignment must be completed by the due date.

The assignment for homework 1 is to create a simple QBasic program that asks questions about the first session of the programming fundamentals class or manual. The format is like a quiz that you would have to take at the end of a course. Please make sure to give the users feedback by using some form of output.

The program requires these four components:

1. The program must work when run.
2. There are comments to show the reader the intentions of the programmer.
3. There is input and output of variables.
4. The program must include the use of IF THEN statements
5. The program contains between 7 and 20 lines of code.

To Get QBasic Files

1. From the MERC, type SST in the address bar.
2. Click **PROGRAMMING FUNDAMENTALS**.
3. Click **FILES** on the left side of the page.
4. Follow the given instructions on the page.

OR

1. Go to http://htmlprod.micron.com/webapps/is/cmpsrv/sst/sstportal/apps/Programming/Programmingfun/Programmingfun.htm
2. Click **FILES** on the left side of the page.
3. Follow the given instructions on the page.

Saving Homework

To save your homework #1 while in QBasic:

1. Click **FILE**.
2. Click **SAVE AS**.
3. Using the scroll bar on the right, scroll down to **[-H-]** drive.
4. Double-click to drill down each step to **MTI** > **MSI** > **IS** > **SMCMIKLE > PROGRAMMING**.
   Note: To go back up a directory, double-click the "**..**" at the top of the list.
5. Enter the program name, which should be **username.BAS**.
   Warning: The program name must be 8 characters or less!
6. Click **OK**.

To get more help on QBasic, refer to "Appendix 1 - Additional QBasic Resources" on page 66.

Due Date: On or before the beginning of Session 2.

PROGRAMMING FUNDAMENTALS - SESSION 2

Objectives

Goal

Establish a basic programming foundation for team members who are interested in advancing their programming knowledge and abilities.

Objectives for this Session

After completing this session, the student should be able to:

- Identify and construct basic logic structures, arrays, subroutines, and functions.
- Identify and recite the programming standards at Micron, including the usage of the Rational Unified Process (RUP), code design, storage, security, ownership, and distribution.
- Identify the standard tools common to most programming languages, such as text writers, editors, compilers, debuggers, interpreters, and bytecode.

ADVANCED PROGRAMMING CONCEPTS

Don't be alarmed by the heading of "Advanced Programming Concepts." These concepts are not difficult, nor are the programs. These concepts are based on the fundamentals you learned in Session 1. This section includes logical constructs; fancy new variables called arrays; a way of breaking your program into smaller parts, called subroutines; and, finally, functions to help you create reusable, single-variable subroutines.

Logic Statements (Part Two)

Often, you need some extra commands in your code to process the flow of the computer program. These are called the logic statements or the control commands. These commands can change the direction of the program or evaluate the input from the users. In the previous area we covered IF THEN statements and now we cover four more logic commands; you will find these commands or derivatives of them in most programming languages.

FOR NEXT Statement

FOR NEXT loops are usually used as counters. These counters help the program repeat itself a set number of times.

Using the recipe example, assume that you want to mix the contents of bowl #2 into bowl #'1 at least three times to ensure that it is well blended. You can use the variable *looptimes* to keep track of the number of loops that cycle through (called iterations):

```
1:  FOR looptimes = 1 to 3
2:  GET spoon
3:  MIX contents of bowl #1 into bowl #2
4:  MIX vigorously for 5 minutes
5:  NEXT looptimes
```

When the computer encounters the FOR statement it sets the variable *looptimes* initially to 1; then, it processes the rest of the instructions until it encounters the NEXT statement. When the set of instructions is completed, the flow returns to the top, the variable *looptimes* is set to 2, and the instructions are repeated. The same steps are repeated when the variable is set to 3. When it is mixed for the third time, the NEXT statement checks the value of *looptimes* to verify that it meets the last FOR number. Then it ends the loop and continues with the rest of the program.

The FOR NEXT example repeats the loop in this program, but it is based upon the number of times that the user specifies. To save time, you may suggest to the user that he or she use a number smaller than 20. As an example of better programming practices, you could alter this program to have an IF THEN statement that forces the user to enter a number less than 20 before the program runs.

New Command Summary:

Command	Definition
FOR NEXT	Logic command set to loop the number of NEXT times.
SLEEP	A simple way to pause a program. The number or variable after sleep will set it for how many seconds you would like it to sleep. FOR NEXT loops can also accomplish this task.

Example Program Name: **FORNEXT.BAS**

```
1:  ' FORNEXT example for Programming Fundamentals
2:  '
3:  '
4:  CLS
5:  DIM repeat, looptimes, timesleft
6:  repeat = 5
7:          FOR looptimes = 1 TO repeat
8:                  SLEEP 1
9:                  PRINT "This will repeat itself ";
    repeat; " times"
10:                 PRINT
11:                 timesleft = repeat - looptimes
12:                 PRINT " Only "; timesleft; " more
    times left"
13:                 PRINT
14:         NEXT looptimes
15: END
```

Program Notes:

- Line 4 clears the screen.
- Line 5 defines the variables.
- Line 6 sets the repeat variable to 5 (the number of loops to complete).
- Line 7 sets the FOR NEXT loop to the number of times that the user has specified.
- Line 8 has the computer pause for one second so the user can see the output.
- Line 9 prints the user input.
- Line 11 sets variable timesleft to be the literal value of “repeat” minus the current value of the FOR NEXT loop counter called looptimes.
- Line 12 prints variable timesleft.
- Line 14 repeats the loop as long as there are more looptimes left.

- Line 15 ends the program.

> **Exercise 5:** Alter the program **FORNEXT.BAS** so that the user inputs the repeat variable number. To save on time, you should also limit the entry to a value of 20 or below.

DO WHILE LOOP Statement

In the DO WHILE LOOP—or known in some languages as DO WHILE—the basic logic lets you run several different operations while waiting for an answer. Going back to the recipe example, you might have several batches of cookies that need various baking times, since oven temperatures and sizes can vary. This means that some batches only take 8 minutes of cook time to finish, while others require 10 to 12 minutes of cook time. The condition of the "rawness" is evaluated and the computer knows whether to repeat the loop.

```
1:  DO WHILE (cookies are raw)
2:  Cook at 375 degrees for one minute
3:  LOOP
```

CAUTION: Be careful not to create code like the sample below. If you review this loop carefully, you will notice that it will run continuously, causing an infinite loop—a very bad condition in a program.

```
1:  DO WHILE (cookies are done)
2:  Cook at 375 degrees for one minute
3:  LOOP
```

In the following example program, the DO WHILE LOOP is evaluated at the beginning of the process, and it tells the computer to continuously run a function until you enter a certain key combination. This functionality was originally created for game programming, allowing the computer to continue working on a graphic while the user sent instructions to other parts of the program.

New Command Summary:

Command	Definition
DO WHILE	Runs a continuous loop while the computer is waiting for new input or a change in a condition; loops always need a way out or they will continue forever. DO WHILE evaluates the logic at the beginning of the process.
INKEY$	Much like INPUT, but it only captures (gets from the keyboard) a single keystroke.
CHR$ (n)	This is used to call out a specific key on the keyboard. It is using ASCII code, which can be found in “Appendix 3 - ASCII Character Set” on page 77.

Example Program Name: **DOWHILE1.BAS**

```
1:  ' Do While1
2:  CLS
3:  DIM pause
4:          DO WHILE (INKEY$ <> CHR$(33))
5:                  PRINT "This will print until you
    press the ! key";
6:                          FOR pause = 1 TO 100000: NEXT
    pause
7:          LOOP
8:  PRINT
9:  PRINT "You escaped!!"
10: END
```

Program Notes:

- Line 1 comments the name of the program.
- Line 2 clears the screen.
- Line 3 defines the variable pause.
- Line 4 starts the DO WHILE LOOP with an evaluation of the first run. DO WHILEs do not have to run, but in this case it will continue until you press the <**!**> key.
- Line 5 prints the statement over and over again.

- Line 6 has a seemingly useless FOR NEXT statement using the variable pause. This is a simple way to keep the computer busy for a brief moment so that the text is readable.
- Line 7 loops back up to check on the line 4 start of the DO WHILE.
- Lines 8 and 9 print a statement to the screen when the loop is completed.
- Line 10 ends the program.

> **Exercise 6:** Comment out line 6 of **DOWHILE1.BAS** and run the program to observe the effects. What happens?

The more complex version of **DOWHILE1.BAS**, called **DOWHILE2.BAS**, is available for your reference after you have had a chance to work with more programs.

New Command Summary:

Command	Definition
SPC (n)	Used to set a print point that is "n" number of spaces from the left of the screen.

Example Program Name: **DOWHILE2.BAS**

```
1:  ' Do While2
2:  CLS
3:  DIM location, pause
4:          DO WHILE (INKEY$ <> CHR$(33))
5:                  location = location + 1
6:                          IF location = 40 THEN
7:                          location = 0
8:                          END IF
9:                  PRINT
10:                 PRINT SPC(location); "This will
    print until you press the ! key";
11:                         FOR pause = 1 TO 10000: NEXT
    pause
12:         LOOP
13: PRINT
14: PRINT
15: PRINT "You escaped!!"
16: END
```

Program Notes:

- Line 1 comments the name of the program.
- Line 2 clears the screen.
- Line 3 defines the variables location and pause.

- Line 4 sets up the loop and evaluates the input from the keyboard to see whether anyone has pressed the **<!>** key.
- Line 5 sets the variable location to location +1.
- Line 6 evaluates location, and if location = 40, then it sets it back to 0. This was included so that the message would start back on the left after it reached the right side.
- Lines 9 and 10 print the message about pressing the **<!>** key to the screen for the user, but it prints it at the point where the location variable tells it.
- Line 11 has a seemingly useless FOR NEXT statement using the variable pause. This is a simple way to keep the computer busy for a brief moment so that the text is readable.
- Line 12 repeats the loop.
- Lines 13 through 15 give feedback to the user when he/she has escaped
- Line 16 ends the program.

Exercise 7: Comment out line 11 of **DOWHILE2.BAS** to observe the effects. What happens?

DO LOOP UNTIL Statement

The following DO LOOP UNTIL is evaluated at the end of the process. In the following example, the loop continues until you guess the correct number between 1 and 1000.

New Command Summary:

Command	Definition
DO LOOP UNTIL	Runs a continuous loop until it meets the conditions set by the programmer; loops always need a way out, or they will run forever. DO LOOP UNTIL evaluates the logic at the end of the process and will always run at least once.
RANDOMIZE	Sets the start of the RND random number generator to simulate real random numbers.
RND	Creates a random number.

Example Program Name: **GUESS.BAS**

```
1:  ' Number guessing game
2:  CLS
3:  DIM count, user, guessable
4:  count = 0
```

```
5:  PRINT "Please enter a number as instructed below"
6:  RANDOMIZE
7:  guessable = INT(RND * 1000)
8:  PRINT
9:  PRINT "Number guessing game"
10: PRINT
11: PRINT "I am thinking of a number between 1 and
    1000...."
12: PRINT
13:         DO
14:                 INPUT "What is your guess"; user
15:                 IF user < guessable THEN PRINT
    "Higher"
16:                 IF user > guessable THEN PRINT
    "Lower"
17:                 count = count + 1
18:         LOOP UNTIL (user = guessable)
19: PRINT "You have guessed the computer's number";
    guessable
20: PRINT "Your guess was"; user
21: PRINT
22: PRINT "It took you"; count; " tries"
23: END
```

Program Notes:

- Line 4 sets the count variable to 0. This is used to ensure that the number is starting at 0, just in case count is used in another area of the program.
- Line 5 gives user instructions about the RANDOMIZE function.
- Line 6 starts the random number generator.
- Line 7 creates a variable called guessable and sets it equal to a number between 1 and 1000.
- Lines 8 through 12 give instructions to the user.
- Line 13 starts a DO loop.
- Line 14 asks for a guess.
- Line 15 hints Higher if the user's number is less than the guessable number.
- Line 16 hints Lower if the user's number is more than the guessable number.
- Line 17 increments the count variable by 1 (counting the number of times in the loop).
- Line 18 finishes the loop by evaluating to see if the user's number equals the guessable number. If this is true, then it will leave; if not, it will continue to loop.
- Lines 19 through 22 give the user feedback on his or her attempt and the number of times that it took to guess the number.
- Line 23 ends the program.

TIP: If you get stuck in QBasic in a loop, press the **<CTRL>** + **<BREAK>** keys simultaneously.

Logic Statement Summary

The following table summarizes when to use the various common logic statements:

Command	When to use
IF THEN	To ask a closed-ended question with only two outcomes, such as a decision diamond in a flow chart.
IF THEN ELSE IF	To ask a closed-ended question with many different possible answers; it is important to include all possible answers.
FOR NEXT	To create a loop that is repeated a specific number of times.
DO WHILE	To create a continuous loop that may never get to run. At the beginning of the loop, it evaluates a logic statement; if correct, it continues the program; if not correct, it continues to loop.
DO UNTIL	To create a continuous loop that will always run at least once. At the end of the loop, it evaluates a logic statement; if correct, it continues the program, if not correct, it continues to loop.

Arrays

Our Phake programming language did not directly include arrays, but you can incorporate them. Arrays are best described as variables that hold a list of items. For example, a typical list of ingredients for the cookie recipe example would read:

- flour
- sugar
- butter
- eggs

For this example, "*ingredients*" is the name of the array, and the recipe items you need are added into the new array variable. The array called "*ingredient*s" would look something like this:

ingredients (flour, sugar, butter, eggs)

If you wanted to tell someone what the third item on your list is, you would say that it is butter. Arrays work the same way; the third item in the *ingredients* array is butter. Arrays can be useful to the programmer for listing items, for sorting, or for keeping together variables of a related type.

Now you can apply the array structure to a program.

Example Program Name: **ARRAY1.BAS**

```
1:  ' My Array1 program
2:  '
3:  '
4:  CLS
5:  DIM dayofweek$(3)
6:  PRINT "What day of the week was yesterday": INPUT
    dayofweek$(1)
7:  PRINT "What day of the week is today": INPUT
    dayofweek$(2)
8:  PRINT "What day of the week is tomorrow": INPUT
    dayofweek$(3)
9:  PRINT
10: PRINT "THANK YOU!"
11: PRINT
12: PRINT "You entered: "; dayofweek$(1); ", ";
    dayofweek$(2); ", and "; dayofweek$(3)
13: END
```

Program Notes:

- Lines 1 to 3 are the comments.
- Line 4 clears the screen.
- Line 5 sets a new array variable called “dayofweek” to three positions.
- Line 6 asks for user input for yesterday’s day.
- Line 7 asks for user input for today’s day.
- Line 8 asks for user input for tomorrow’s day.
- Line 10 thanks the user.
- Line 12 sends the entered information back to the user.
- Line 13 ends the program.

Exercise 8: Modify **ARRAY1.BAS** so the user also enters the day after tomorrow (2 days from today). Be sure to change the number of items in the array and modify the printed text accordingly.

New Command and Syntax Summary:

Syntax	Definition
(n)	A number used after an array to designate the placement number of the item in the array.

TIP: Be careful with array types and names in different programming languages. Some arrays consider number (1) to be the first in the list, and others start with number (0). While an array can start at any number the

programmer would like, you should only change it if it helps explain the code or if it is commented. Otherwise, stick to the standards for the language with which you are working.

The next example uses an array that is predefined by the program so it can ask one question and produce the same results.

Example Program Name: **ARRAY2.BAS**

```
 1:  ' My Array2 program
 2:  '
 3:  '
 4:  CLS
 5:  DIM dayofweek$(7), inputday
 6:  dayofweek$(1) = "Sunday"
 7:  dayofweek$(2) = "Monday"
 8:  dayofweek$(3) = "Tuesday"
 9:  dayofweek$(4) = "Wednesday"
10: dayofweek$(5) = "Thursday"
11: dayofweek$(6) = "Friday"
12: dayofweek$(7) = "Saturday"
13: PRINT "Enter the number that corresponds to the day
    of the week"
14: PRINT
15: PRINT "Sunday is        1"
16: PRINT "Monday is        2"
17: PRINT "Tuesday is       3"
18: PRINT "Wednesday is     4"
19: PRINT "Thursday is      5"
20: PRINT "Friday is        6"
21: PRINT "Saturday is      7"
22: PRINT
23: INPUT inputday
24: PRINT
25: PRINT "THANK YOU!"
26: PRINT
27: PRINT "Yesterday was "; dayofweek$(inputday - 1);
    ", Today is "; dayofweek$(inputday); ", and
    Tomorrow is "; dayofweek$(inputday + 1)
28: END
```

Program Notes:

- Lines 1 to 3 comment the program.
- Line 4 clears the screen.
- Line 5 sets up a new 7-position array called dayofweek and defines a variable called inputday.
- Lines 6 to 12 set the dayofweek slots in the dayofweek array to the appropriate days.

- Lines 13 to 21 give the user the available days of the week and their corresponding numbers.
- Lines 22 to 23 asks for the user input.
- Line 25 is polite.
- Line 27 gives the output like the previous program, only now the computer knows the right dates (but only if the right date is entered!).

Exercise 9: In **ARRAY2.BAS**, if the user indicates that today is either Saturday or Sunday, the program will not function correctly because 0 (1 - 1) and 8 (7 + 1) are not defined in the array. Improve the logic of the program so the program will run correctly.

TIP: Some programming languages save you some work by including predefined days-of-the-week or days-of-the-month arrays.

Subroutines

Subroutines simply take repeated or commonly used code and set it aside from the main code. This makes editing easy; if you want to change every instance of a repeated set of instructions in your code, you simply change it once in the subroutine window. In QBasic, subroutines are saved when the main program is saved.

New Command and Syntax Summary:

Command	Definition
DECLARE SUB subname (variables)	The declaration does two things: 1) sets up a sub with subname and 2) sets the variables that will be used by both the main program and the subroutine.
SUB	Starts the subroutine.
CALL	Runs the program or subroutine name that follows.
END SUB	Returns the flow of the program back to where it left to go to the subroutine. This is much more predictable than earlier GOTO statements.

Syntax	Definition
DECLARE SUB subname (variables)	The variables declared in the variables section have to be the same type as the variables used with the SUB command.

Review the example below to see the usage. Notice instead of using DIM to define the variables, you use DECLARE, which makes the variables available not only to the main program but also to the subroutines.

Example Program Name: **SUBROUTE.BAS**

```
1:  DECLARE SUB combo (name$, message$)
2:  'A simple example of subroutine
3:  '
4:  '
5:  CLS
6:  PRINT "This is a sample of how subroutines work"
7:  PRINT
8:  PRINT
9:  PRINT "Please enter your name": INPUT name$
10: CALL combo(name$, message$)
11: PRINT message$
12: END
```

The following is the subroutine of the **SUBROUTE.BAS** program.

```
1:  SUB combo (name$, message$)
2:          IF name$ = "Shaun" THEN
3:          message$ = name$ + " is a great instructor"
4:         ELSE message$ = name$ + " is a good student"
5:          END IF
6:  END SUB
```

Program Notes:

- Line 1 declares the subroutine called combo and states that this subroutine has two variables called name and message.
- Lines 2 to 4 are comments.
- Line 5 clears the screen.
- Line 6 prints a message saying this is a sample of how subroutines work.
- Line 9 asks the user to input his or her name.
- Line 10 calls the subroutine (or goes to the subroutine). The program will return to this same spot when it receives the END SUB command.

- Line 1 of the SUB sets up the two variables to be used by both the main program and the subroutine.
- Line 2 checks to see whether the name string variable is equal to Shaun (comparison).
- Line 3 says if the name is Shaun, then the program sets the message variable to the name string variable plus *is a great instructor* (concatenate).
- Line 4 says if the name is not Shaun, then the program sets the message variable to the name string variable plus *is a good student.*
- Line 5 of the SUB ends the IF statements.

- Line 6 ends the subroutine and returns to the place in the main program where it originally branched to the subroutine (in this case, Line 10).

> **Exercise 10:** Add a new question in the main program of **SUBROUTE.BAS** that asks the user to input a description of the person whose name was entered. Then pass that variable information to the subroutine for concatenation.

The Mad Lib program that we used earlier has been converted to provide another example of the use of subroutines.

Example Program Name: **MADLIB3.BAS**

```
1:  DECLARE SUB gender (them$, pronoun$)
2:  '  This is my program called madlib3.bas
3:  '  I wrote it for my Programming Fundamentals Class
4:  '
5:  CLS
6:  DIM actioning$, animal$, someone$, funny$, farm$,
    bodypart$
7:  PRINT "Here is a little quiz.  Please use funny
    words,"
8:  PRINT "because they make this more interesting."
9:  PRINT
10: PRINT
11: PRINT "Enter an action word ending in ING": INPUT
    actioning$
12: PRINT "Enter an exotic animal name": INPUT animal$
13: PRINT "Enter your friend's name": INPUT someone$
14: PRINT "What gender is, "; someone$; "? Enter Male
    or Female": INPUT them$
15: CALL gender(them$, pronoun$)
16: PRINT "Enter a funny sound": INPUT funny$
17: PRINT "Enter a farm animal": INPUT farm$
18: PRINT "Enter a body part": INPUT bodypart$
19: CLS
20: PRINT
21: PRINT
22: PRINT "One day many years ago, "; someone$; " was
    "; actioning$; " when suddenly there was a(an) ";
    funny$; " sound, much like a(an) "; animal$; ".  ";
    "Frightened and scared "; someone$; " tripped over
    a "; farm$; "."; " In haste to get away, ";
    someone$; " broke "; pronoun$; " "; bodypart$; "."
23: END
```

The following is the subroutine of the **MADLIB3.BAS** program.

```
1:  SUB gender (them$, pronoun$)
2:       IF them$ = "Male" THEN
```

```
3:       pronoun$ = "his"
4:       ELSEIF them$ = "Female" THEN
5:       pronoun$ = "her"
6:       ELSEIF them$ = "" THEN
7:       pronoun$ = "his or her"
8:       ELSE
9:      PRINT "BAD ENTRY, ERROR.": pronoun$ = "his or
   her"
10:      END IF
11: END SUB
```

Creating a Subroutine

To create a subroutine

1. Save your program.
2. Click **EDIT**, and then select **NEW SUB**.
3. Enter the name of the subroutine, and then click **OK.**
4. On the same line as the name of the subroutine, specify the name of the variables you would like to make available for both the main program and the subprogram. Enclose these variables in parentheses ().
5. Write your subroutine.
6. **SAVE** your work.

Within the program, use the **<F2>** key to switch back to the main program area. Use the Call command to access the subroutine when needed.

Functions

A function is simply a single variable subroutine that you can teach the computer to run when you need a repeated action. The advantage of using a function is that you can change or upgrade the function without changing the main program. For example, you can create a function to calculate your after-tax paycheck when Uncle Sam is finished with your original gross income. Then when you evaluate salary levels, you will know your take-home pay.

Note: The calculations in the following example are just for fun and do not reflect actual payroll information.

New Command and Syntax Summary:

Command	Definition
DECLARE Function (Variable)	Declares the function so that the program can find it later when called.
FUNCTION	Defines the start of the function area and the function name.
END FUNCTION	Ends the Function and returns to the call point.

Syntax	Definition
Function (variable)	Verifies that the variable type in the function matches the function itself.

Example Program Name: **FUNCTION.BAS**

```
1:  DECLARE FUNCTION Taxes! (number)
2:  'Function Program
3:  '
4:  '
5:  CLS
6:  ' Constant hours in year based on standard hours
7:  hoursinyear = 2080
8:  PRINT "Are you paid hourly or salary? Input H or
    S": INPUT Howpaid$
9:  IF Howpaid$ = "S" THEN
10:         PRINT "Input your annual salary": INPUT
    wages
11:         hourly = wages / hoursinyear
12: END IF
13: IF Howpaid$ = "H" THEN
14:         PRINT "Input your hourly wage": INPUT
    hourly
15:         wages = hourly * hoursinyear
16: END IF
17: PRINT
18: PRINT "Your hourly wage before taxes is "; hourly
19: PRINT "Your annual salary before taxes is "; wages
20: PRINT
21: PRINT "Your hourly wage after taxes is ";
    Taxes(hourly)
22: PRINT "Your annual salary after taxes is ";
    Taxes(wages)
```

The following is the breakdown of the function in the **FUNCTION.BAS** program.

```
1:  FUNCTION Taxes (number)
2:  incomeTaxes = number * .15
3:  Taxes = number - incomeTaxes
4:  END FUNCTION
```

Program Notes:

- Line 1 declares the function.
- Line 6 comments about the constant.

- Line 7 sets a constant called hoursinyear.
- Line 8 asks the user to input pay type.
- Line 9 asks a question only if the user's income is paid in salary.
- Line 13 asks a question only if the user's income is paid hourly.
- Lines 18 and 19 print the current salary information.
- Lines 21 and 22 print the salary after taxes.

- Line 1 of the Function sets up the function called Taxes.
- Line 2 of the Function will take the number in Taxes and acts upon it by multiplying it by 15%.
- Line 3 of the Function sets taxes to the number - the incomeTaxes.
- Line 4 of the Function ends the function.

Creating a Function

To create a function:

1. Save your program.
2. Click **EDIT**, and then select **NEW FUNCTION**.
3. Enter the name of the function. On the same line as the name of the function, specify the name of the variable that you would like to make available for both the main program and the subfunction. Enclose that variable in parentheses (), and then click **OK**.
4. Write your function.
5. **SAVE** your work.

Within the program, press the **<F2>** key to switch between the main program and your functions.

Subroutine and Function Summary

The table summarizes when to use subroutines or functions:

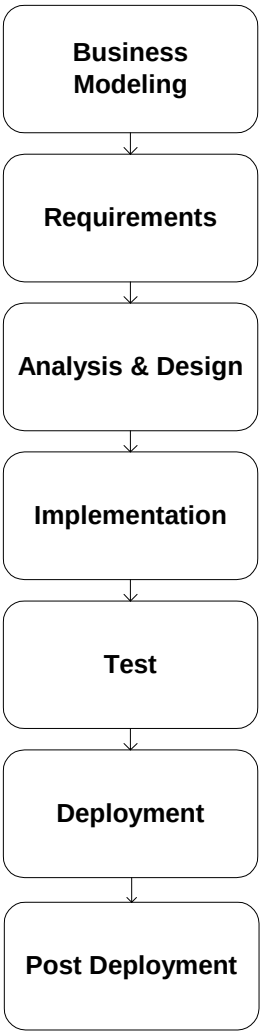
Command	When to use
SUBROUTINE	Used when you have more than one variable that you would like to pass back and forth from the main program and the subroutine.
FUNCTION	Used when you only have a single variable that you would like to pass to a subfunction and have a series of actions (commands) applied to it.

PROGRAMMING STANDARDS

Computer programming is more than just writing code and running the programs that you have created. It includes standards for developing and designing code, as well as methods for securing, saving, and distributing that code. The topics discussed in this section will prevent you from making the same mistakes that others have made in the past. Keep in mind that this is only a small portion of the "required" knowledge for Micron programmers. If you continue to program, it is highly recommended that you study the information provided in the Appendices, consult with Information Systems web sites, and review the books listed as references.

Developing using Micron's Standard

Micron uses the Rational Unified Process (RUP) to help programmers develop an application that meets Micron standards. RUP consists of a series of steps and helps ensure that we are producing high-quality software that meets our end-users' needs, within a predictable schedule and budget. Here is a summary of the RUP stages:



Business Modeling is the "investigation" stage where you observe and document how the current business process works.

Requirements is the "clarification" stage where you work with your customers to verify their expectations and confirm that you can build the product.

Analysis & Design is the "write it down" stage where you take the modeling and requirements and actually develop a comprehensive plan.

Implementation is the "just do it" stage where you take the plans and build the product.

Test is the "break it" stage where you verify that your product meets all of your customers' needs.

Deployment is the "release" stage where you give the products to your customers.

Post Deployment is the "improvements" stage where you keep track of bugs, enhancements, and customer requests.

For more information on the Rational Unified Process:

1. Launch the **MERC**.
2. In the **Address** bar, type **SQA** and then press <**Enter**>. The SQA home page displays.
3. Click the plus sign (+) next to **3rd Party Applications**, and then click **Rational Unified Process**. The RUP home page displays.

Designing Code

The goal of every programmer should be to develop programs that are well designed and accomplish great tasks. Many programmers agree that the following recommendations are essential programming standards. According to Micron's Software Quality Assurance group, the most important of these standards is readability.

Commenting Your Code

Other programmers cannot capture your intentions and thought processes by simply looking at your source code. It is extremely difficult to understand, interpret, and navigate through code and statements that have not been commented. Adding comments before every section of your project code is helpful when others read your work or when you need to retrace your work. If you are unsure about the detail of the comments, ask someone unrelated to the project to read the code; if he or she can understand it with little or no explanation, you have been successful.

Ensuring Readability

Readability refers to the structure and general format of the code that you write and is considered the most important programming standard at Micron. In many programming languages you can write many instructions together, write complex nesting structures, and make calls to external files and programs. In each of these cases, you should keep readability foremost in your mind. Using correct line structure, indenting code, and commenting the code is expected at Micron. How you implement these techniques will directly impact how you are evaluated as a programmer.

Writing Efficient Code

Efficient code is best defined as using the fewest number of steps to accomplish a computing task or making the most efficient use of resources. Shorter and more efficient code saves hard drive space and requires fewer computing resources to run. It is much easier to find errors and problems in small, easy-to-follow programs. Be careful to balance the readability of the code with the required performance level; with today's computing power and the fact that many people read Micron code, make readability a higher priority.

Handling Errors

Error handling is anticipating user responses by brainstorming all possible outcomes and then generating code or subroutines to match those possibilities. Sometimes the ambiguities or other interpretations of the interface cannot be seen by the programmer because he or she makes assumptions or is too close to the project. To determine whether a choice is ambiguous in nature, ask others for help with this process. See “Appendix 4 - Vocabulary Exercise Answers” on page 78 for more information on testers and testing processes.

Trapping Errors

Another approach to handling errors is to use error trapping. With this method, you trap errors generated by your program to a holding bin or file so that you can review and fix them later. Tracking errors and problems that have occurred in the past will improve not only the program but also the programmer. The use of log files, trace files, and archive files help facilitate this process. For more information on errors and error trapping, see “What’s Next?” on page 88.

Saving Your Work

ALWAYS SAVE YOUR WORK. **ALWAYS SAVE YOUR WORK**.

ALWAYS SAVE YOUR WORK. **ALWAYS SAVE YOUR WORK**.

Repeat this statement numerous times, so you are sure to remember. When working with computers, it is critical that you regularly save your work because there is always the possibility of power loss, corruption, or other circumstances beyond your control.

Equally important is **where** you save the work. At Micron, you must save your files to a network drive. The most likely drive for this use is the F: drive, which is accessible only by you. It is backed up nightly and can be retrieved by the Support Center if deleted or corrupted. You should not save programs or related files on a floppy drive (A:) or on your local hard drive (C:). Neither of these volatile and insecure places of storage are acceptable long-term storage locations for Micron property. You may find it necessary to store files on the C: drive temporarily for a large work in progress, but you should only do this if you have automated a method for making a nightly copy to a network server.

When creating files on the F: drive, first create a folder that contains all the project files for the program on which you are working. Many programming languages create multiple files that are required for running the code. For ease of navigation, name the folder exactly the same as the project.

If you have any work saved to a common network drive, such as G:, it is imperative to establish strict security permissions. Copying a program file out to a public drive can subject yourself and Micron to many different dangers. If you have questions about creating permissions and establishing safe zones, consult the Information Security Team in Information Systems.

PVCS at Micron

To protect our valuable software assets, Micron’s mission-critical programs are stored in a system called Project Version Control System (PVCS). PVCS Version Manager serves as the primary repository for all the source files used to create production applications.

These files can include specifications, source code, and make files, as well as binaries, such as bitmaps, application-specific DLLs, and special controls.

PVCS provides Micron the ability to:

- Store multiple revisions of each source file in the archive
- Lock an archive so that it can be updated by only one developer at a time
- Tie specific revisions of each source file to specific releases of an application
- Automatically maintain an audit trail of changes to all of the files under version control

PVCS should be used:

- To store revisions of **all** production source files
- To coordinate development between two or more developers working on the same project
- To store a revision of a project under development whenever any major changes are made

To access more information about PVCS:

1. Launch the MERC.
2. Type **SQA** into the Address field, and then press **<ENTER>**.
3. Click the plus sign (**+**) next to **3RD PARTY APPLICATIONS**.
4. Click **PVCS VERSION MANAGER**.

You can also send an e-mail message to the SQA team at *sqa@micron.com*.

Securing Your Work

After your code is in production, the program security that you use will affect not only your access but also the access of anyone else in the enterprise. There are three issues to consider when setting security on your programs.

- As you develop your code, test the authentication from a developer's account, an anonymous account, and from a user's account. Authentication may require a different syntax or may expose authentication issues that would otherwise stay hidden until after the code has been released.
- Do not enforce complex or unique passwords during the development stages because that may become a nuisance. These restrictions can be added later.
- Do not use administrative (admin) accounts when developing code. Admin accounts have no access restriction, and the code can do anything at anytime.

Source Code Ownership

The programming code written at Micron, or for Micron, belongs to Micron. You may be the author, but Micron is the owner. Many programs that you use today are critical to the operations of the company. The loss of this data or loss of the process by which it is created can cost the company millions of dollars. Code developed at Micron or for Micron cannot leave Micron, even if you leave the company. If there is ever a doubt, consult your supervisor or Information Security immediately.

Distribution of Your Work

With the advances of computer programming and networking, creating and distributing programs have become much easier. One of the greatest responsibilities of a programmer is to distribute programs in an organized and beneficial way. Whole teams of IS professionals have been developed to ensure that computer programs are developed and distributed correctly. The Developer Tools and Languages (DTL) and Software Quality Assurance (SQA) teams have volumes of information on developing better code.

- The Software Quality Assurance (SQA) group is chartered with overseeing development, training, and administration of software changes on all platforms. This includes producing procedures, tools, and processes that will enforce a strong commitment to software quality among the development community. While this team is **not** responsible for the overall software quality here at Micron, they are responsible for ensuring that developers produce quality software. To visit the SQA group web site, type **SQA** in the Address field of the MERC.
- The Developer Tools and Languages (DTL) group provides quality support and training for the many different programming languages and development tools used by Micron's IS developer community within the NT, Unix, and OpenVMS environments. If you are having problems with a program written in Visual Basic, C, C++, Perl, or other supported programming languages at Micron and cannot find a documented solution, you can contact the DTL group for assistance in solving your problem. To visit the DTL group web site, type **DTL** in the Address field of the MERC.

PROGRAMMING TOOLS

To start programming in an authentic programming language, you first need to have the tools of the computer programmer. Each programming language has its own tools to aid you in the programming process, but they all use the same basic models.

Text Editors

The most basic way to write a program is to use a text editor. These editors create CUI-based text files, and they do not have any tools to aid the programmer. Many people who create web pages, or develop for PERL, use only text editors because the resulting files are universal to all computer systems. One of the first text editors that came with DOS was ED (and later EDLIN), and Unix's editor is called *vi*. Today, all Microsoft operating systems come with a simple text editor called Notepad.

Code Editors

Code editors are one step above simple text writers because they take a simple word processor and add some specific tools related to a specific programming language. Code editors include tools that help the programmer, such as syntax checkers, help files, and debugging tools (described in QBasic).

When you purchase a computer programming package, a code editor is usually included. In some cases the code editor can be obtained for free.

Debuggers

Removing the errors that can occur in a computer program is called debugging. In programming, there are many points of failure: your source code, the editor you are using, the compiler, the compiled program, the dependency libraries, and interaction with other programs. While code editors and compilers often warn you of specific error types, the quest to create a bug-free (error-free) program never ends. For further information on errors, consult "Debugging" on page 72.

Common ways to debug a program include:

- **Step through the program**: Use the debugger to find logic errors by configuring the editor to show you each error as it occurs in the program.
- **Use breakpoints**: Instead of stepping throughout the entire program, use breakpoints to step through just a specific portion of the code.

Compilers

A compiler reads the source code that you have written with a text or code editor and translates it to machine code. Because computers only talk in 1s and 0s and all computer languages are written in words, the compiler converts those words to a language that the computer understands. This is illustrated below.

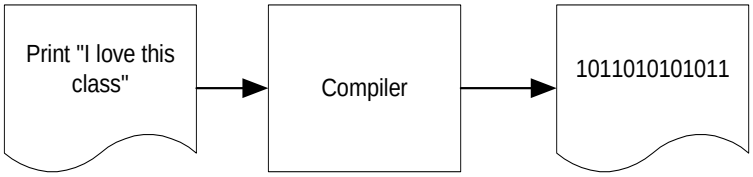


Figure 8: Diagram of Compiler

After a program has been compiled, it becomes a stand-alone executable file that can be run on any computer with which the compiler is compatible.

> **Note:** If you want to build a program that will run on all platforms, you either have to create multiple copies for different compilers or use new programming languages like PERL or JAVA.

Interpreters

Interpreters, like compilers, translate text information that you have developed into machine language; however, when the interpreter translates the code, it processes the text information one line at a time into system memory. After it is interpreted, the program is very fast, but when you turn the computer off, the information is lost. Interpreters work very well for web-based programming like JAVA Script and VB Script, but they are not very effective for general programming. QBasic is an example of both an editor and an interpreted language.

Bytecode

The bytecode approach to compiling is a bit more interesting than the previous examples. Instead of writing a program and then translating it to machine code for the computer, you now write your program and translate it to a universal code. This universal code can be sent to other users, who can then compile it for their computers. It would become very problematic to learn how to compile a program each time, so computer program manufacturers, such as Microsoft and Netscape (AOL), have built the translators (just-in-time compilers) into their Internet browsers, such as Internet Explorer or Netscape Navigator.

The best-known language today that uses the bytecode (virtual machine) approach is JAVA. Rather than being interpreted one instruction at a time, JAVA bytecode can be recompiled on each particular system platform by a just-in-time compiler. Usually, this enables the JAVA program to run faster.

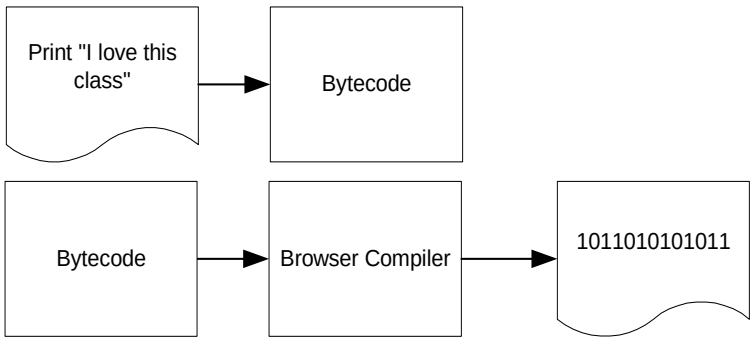


Figure 9: Diagram of Bytecode Compiler

PROGRAMMING AT MICRON

	Program	Description	Primary Tool	Example
Procedural Programming Languages	C	C is the first procedural language that falls into the portable assembler class. It can directly manipulate hardware like machine code or assembly, but it is a high-level program. C is difficult to learn – it was written by programmers for programmers.	Editor	All Windows, Linux, Netscape, Quicken
	C++	C++ expands on the functionality of the C language with a super set of instructions. It can uniquely use C code within C++ programs.	Editor	All Windows, Linux, Netscape, Quicken
	Cobol (Common Business Oriented Language)	Cobol was the first high-level programming language that was widely used for business applications. Since Cobol continues to run huge, mission-critical applications, many companies are continuing with Cobol support until the applications can be rewritten in modern languages.	Editor	
	Fortran (Formula Translator)	Fortran was designed for use by engineers, mathematicians, and creators of scientific algorithms. Fortran is one of the original programming languages and many applications were written with it.	Editor	OPERCERT
	PERL (Practical Extraction and Reporting Language)	PERL is a C-based procedural language and can work from virtually any platform without any changes to the code. PERL defies normal programming conventions in that it can be used as either a procedural or an object-oriented language, and PERL also has bytecode capabilities (see “Bytecode” on page 62).	Text Writer / Editor	Data extraction, systems management
Object-Oriented Programming (OOP) Languages	JAVA and JAVA Script	JAVA was developed by Sun Microsystems. JAVA’s design can work exclusively from networks, instead of from single workstations or servers. It is similar to C++ but easier to use. JAVA also is known for its bytecode capability (see “Bytecode” on page 62).	Text Writer / Editor	Web page pull-down menus
	Visual Basic (VB)	VB allows a user to create GUI applications. This language is an extension of the original BASIC; it encompasses many of the same functions and commands.	Visual Editor	Ship Doc, FAB applications on the PCs
	Visual C++	Visual C++ takes the C++ language and expands it to be object-oriented with a GUI interface.	Visual Editor	Windows, front-end applications
Markup Languages	HTML (Hypertext Markup Language)	HTML is a text-based language that describes web pages to an interpreter (browser), which then displays the page to the user.	Text Writer / Editor	Web pages
	XML (Extensible Markup Language)	XML is a way to extract data in a universal format regardless of platform. This expands the use of data models without having to worry about inter-operability.	Text Writer / Editor	Electronic Data Interchange

VOCABULARY EXERCISE

Match the vocabulary items on the top with their correct definitions on the bottom.

Compiler	**Debugger**	**Loop**	**Binary**	**Variable**
Syntax	**String Variable**	**Function**	**Constant**	**Flow Chart**
Subroutine	**Array**	**Phake**	**GUI**	**Literal**

_____ 1. A variable that holds a list of items that share the same properties or need to be grouped together.

_____ 2. A container that holds information.

_____ 3. Used to describe the interface where the actions that drive the computer are all based upon the user's input through visual navigation.

_____ 4. The name for the value that is contained within a variable.

_____ 5. A pseudo-code programming language invented for this class to illustrate programming logic.

_____ 6. A modeling tool used to best simulate computer logic or step-by-step instructions. It is very useful when there are multiple ways to produce an outcome.

_____ 7. The exact order or logical structure of a programming command. This word is used to denote the order in which a language is grammatically correct.

_____ 8. A variable that holds a specific piece of information throughout the program.

_____ 9. Used when you have only one variable that you need to pass between the main program and the sub program.

_____ 10. A tool to aid the programmer in removing all of the possible errors that can occur in computer programming.

_____ 11. Used when you have more than one variable that you need to pass between the main program and the sub program.

_____ 12. A programming tool that converts source code written in a high level language into machine code.

_____ 13. A continuous logic construct that evaluates a logic statement; if correct it continues the program, if not correct it continues to cycle.

_____ 14. A type of variable that holds character based information.

_____ 15. A numbering system that computers use to communicate, and consists of just two unique digits, "1s" and "0s."

HOMEWORK 2

Before you can receive credit for this class, there is a final homework assignment that is due two weeks after the second session.

The assignment for homework 2 is to expand upon the homework from last week by creating a more sophisticated version of the quiz on the contents of the Programming Fundamentals course. Remember, in programming there is no restriction on how you accomplish this task as long as you meet the requirements.

The program requires these seven components:

1. The program must work when run.
2. There are comments to show the reader the intentions of the programmer.
3. There is input and output of variables.
4. The program contains both numerical variables and string variables.
5. You grade the input of the user, and give the output as to his or her progress.
6. The program includes at least two of these advanced features: logic statements, arrays, subroutines, or functions.
7. The program contains between 20 and 50 lines of code.

To save your homework #2 in QBasic:

1. Click **FILE**.
2. Click **SAVE AS**.
3. Using the scroll bar on the right, scroll down to **[-H-]** drive.
4. Double-click to drill down each step to **MTI** > **MSI** > **IS** > **SMCMIKLE** > **PROGRAMMING**.
   Note: To go back up a directory, double-click the "**..**" at the top of the list.
5. Enter the program name, which should be **usernam2.BAS** (be sure to include the 2 so that you do not copy over your first homework).
   Warning: This must be 8 characters or less!
6. Click **OK**.

To get more help on QBasic, refer to "Appendix 1 - Additional QBasic Resources" on page 66.

Due Date: Within two (2) weeks of class.

APPENDIX 1 - ADDITIONAL QBASIC RESOURCES

Accessing Help in QBasic

Online help is available in QBasic to aid the programmer with various tasks.

To access help in QBasic:

1. In the standard Edit mode, press <**SHIFT**> + <**F1**>. (In the Introduction Screen, press <**F1**>.) The QBasic help screen displays, as illustrated below.

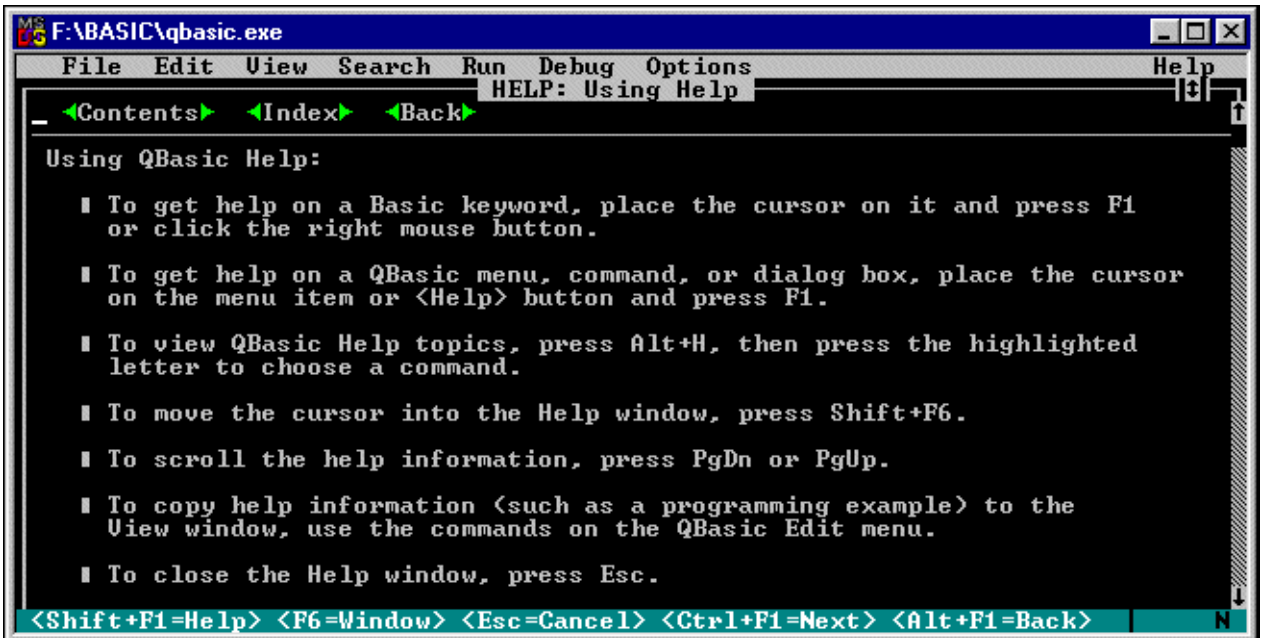


Figure 10: QBasic Help Screen

2. Review the instructions for using QBasic help.
3. Double-click on Contents to display the table of contents for the help section, as illustrated below.

Figure 11: QBasic Help Screen - Table of Contents

To search for a specific topic in the index:

1. Double-click **INDEX.** The Index screen displays as illustrated below.

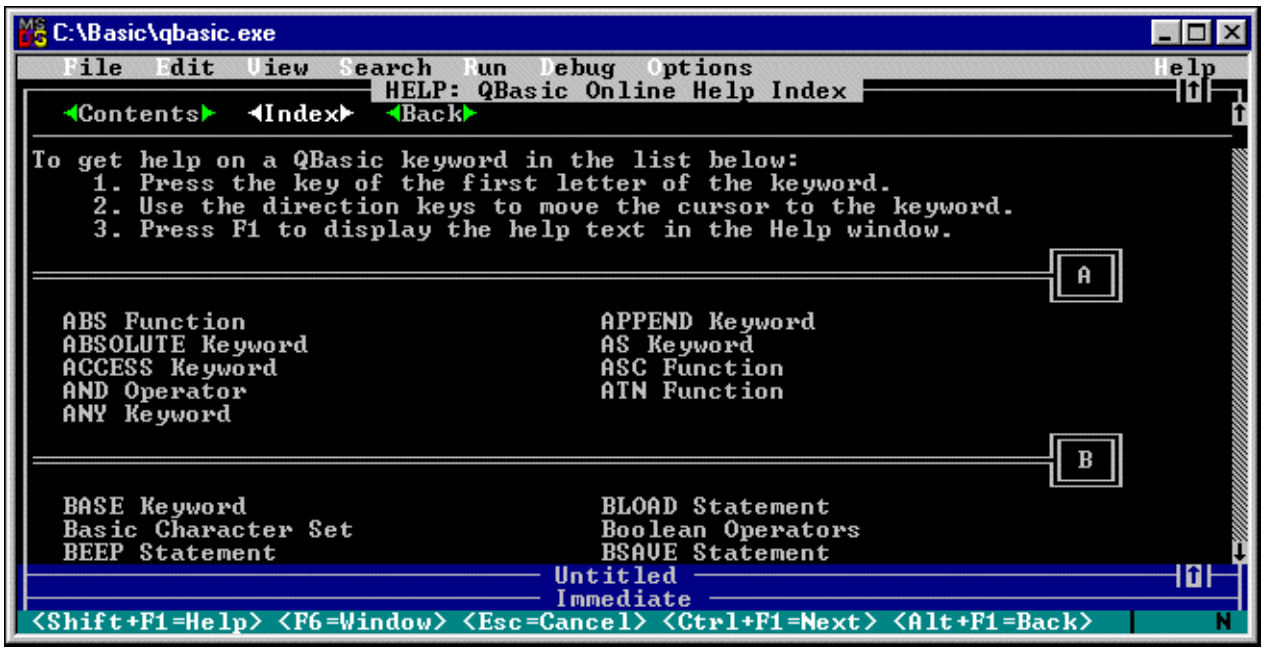


Figure 12: QBasic Help Screen - Index

2. Type the first letter of the command for which you want to search.
3. When the list displays, double-click on the command for the related help file topic. This help file assumes basic knowledge of the program, but does give practical examples of use.

Running QBasic in Full Screen Mode

By default, QBasic runs in a small window. This restore window display runs the QBasic program like a standard Windows application (which it is not), and it consumes a large amount of computing resources. Switching to full screen mode will dedicate your system to working on QBasic alone. This will, however, step your computer back to a basic CUI hybrid platform without the aid of standard Windows icons or menu bars.

To run QBasic in full screen mode:

Press **<ALT>** + **<ENTER>** to switch from QBasic full screen mode and restore window display.

To return to Windows programs:

Press **<ALT>** + **<TAB>** to switch between QBasic and standard Windows programs.

Graphics

Because QBasic graphics were created using languages that were limited to basic programming instructions, they are nothing like what you see today on Nintendo 64 or PS/2. Modern programming languages can access millions of colors and create complex simulations of real environments, but you have to start somewhere.

New Command Summary:

Command	Definition
COLOR	Determines the foreground and background colors.
INT	Changes the number from a decimal to an integer.
LINE	Draws a line.
CIRCLE	Draws a circle.

Example Program Name: **PRETTY.BAS**

```
1:  CLS
2:  SCREEN 9
3:  RANDOMIZE
4:  CLS
5:  c1 = INT(RND * 15)
6:  c2 = INT(RND * 15)
7:  COLOR c1, c2
8:  FOR Z = 1 TO 75
9:  x = INT(RND * 800)
10: y = INT(RND * 800)
11: x2 = INT(RND * 3)
12: y2 = INT(RND * 3)
13: x3 = INT(RND * 200)
14: LINE (x, y)-(1 * x2, 1 * y2)
15: CIRCLE (x, y), x3, c1 + 1
16: NEXT Z
17: END
```

Program Notes:

- Line 2 sets the screen to a VGA mode.
- Line 3 sets the random number generator.
- Line 4 clears the screen.
- Line 5 sets the variable c1 to a number from 1 to 15.
- Line 6 sets the variable c2 to a number from 1 to 15.
- Line 7 sets the color scheme to c1 and c2 values.
- Line 8 sets the FOR NEXT loop to 75 times.
- Line 9 sets x at a random number from 1 to 800.
- Line 10 sets y at a random number from 1 to 800.
- Line 11 sets x2 at a random number from 1 to 3.
- Line 12 sets y2 at a random number from 1 to 3.
- Line 13 sets x3 at a random number from 1 to 200.
- Line 14 creates a line.

- Line 15 creates a circle.
- Line 16 completes the FOR NEXT loop for 75 times.

Sound

QBasic is able to handle sound through the PC speaker. Please note that your computer must have a basic internal speaker, but no sound card is required. Here is a simple noise-making program that increases sound, and then decreases the sound. It starts at a frequency roughly equal to the musical note A, then works up at least an octave, and then returns to A.

New Command Summary:

Command	Definition
SOUND	Produces sound on the PC speaker at a tone frequency.
STEP	Used with the FOR NEXT command to have the FOR NEXT loop to skip by the number stated. In this example it jumps by 5s through the number sequence.

Example Program Name: **SOUND.BAS**

```
1:  CLS
2:  PRINT "Hold your ears...."
3:  FOR note = 450 TO 750 STEP 5
4:          SOUND note, 1
5:          SOUND 800 - note, 1
6:  NEXT note
7:
8:  FOR note = 700 TO 450 STEP -5
9:          SOUND note, 1
10:        SOUND 750 + note, 1
11: NEXT note
12: END
```

Program Notes:

- Line 2 gives the user a warning.
- Lines 3 to 6 set up a FOR NEXT loop that starts the process of STEPping through 450 to 750 cycles.
- Lines 8 to 11 set up a FOR NEXT loop that reverses the process of STEPping through 750 to 450 cycles.

Instead of using the SOUND command, you can specify musical notes with the PLAY command:

New Command Summary:

Command	Definition
PLAY	Plays specific notes.

Example Program Name: **PLAY.BAS**

```
1:  CLS
2:  PLAY "L4 C"
3:  PLAY "L8 B"
4:  PLAY "L4 AGFGAF"
5:  END
```

Program Notes:

- Line 2 plays the C note for a length of 4.
- Line 3 plays the B note for a length of 8.
- Line 4 plays the notes AGFGAF for a length of 4.

QBasic Commands

Command	Definition	Command	Definition
’	This little mark replaces the REM statement in later versions of BASIC.	END FUNCTION	This command ends the Function and returns to the call point.
CALL	This is used to run the program or subroutine name that follows.	END IF	This command stops the IF THEN ELSE IF string of commands. This has helped replace the GOTO command.
CHR$(n)	This is used to call out a specific key on the keyboard. It is using ASCII code, which can be found in “Appendix 3 - ASCII Character Set” on page 77.	END SUB	This command returns the flow of the program back to where it had left to go to the subroutine. This command is much more predictable than earlier GOTO statements.
CIRCLE	This command draws a circle.	ELSE	The final ELSE IF is designated with a single ELSE statement to signal the end of the logic.
CLS	This command clears the screen of all writing.	ELSE IF	This command allows you to continue with an IF THEN statement by giving you more choices.
COLOR	This command determines the foreground and background colors.	FUNCTION	This command defines the start of the function area and function name.
DECLARE Function (Variable)	This command declares the function so that the program can find it later when called.	FOR NEXT	This command is the Logic Command set to loop the number of NEXT times.

Command	Definition	Command	Definition
DECLARE SUB subname (variables)	When declaring, you are doing two things: 1) setting up a sub with subname and 2) setting the variables that will be used by both the main program and the subroutine.	GOTO	This command moves you to a specific line or location.
DIM	This command creates and defines new variables for use in the program.	IF THEN	IF THEN works on the condition; if the statement is true, then the action that follows will occur.
DO LOOP UNTIL	This command runs a continuous loop until it meets the conditions set by the programmer; loops always need a way out, or they will continue forever. DO LOOP UNTIL evaluates the logic at the end of the process and will always run at least once.	INKEY$	This command is much like INPUT, but it only captures (gets from the keyboard) a single keystroke.
DO WHILE	This command runs a continuous loop while the computer is waiting for new input or a change in a condition; loops always need a way out, or they will continue forever. DO WHILE evaluates the logic at the beginning of the process.	INPUT	This command receives input from the keyboard and assigns it to a variable.
END	This command finishes the program.	INT	This command changes the number from a decimal to an integer.
LCASE$(variable$)	This command changes the string variable$ to all lowercase letters.	SLEEP	This command is a simple way to pause a program. The number or variable after sleep will set it for the number of seconds you would like. FOR NEXT loops can also accomplish this task.
LEFT$(variable$, n)	This command cuts the string variable$ by (n) number of characters from the left.	SOUND	This command produces sound on the PC speaker at a tone frequency.
LINE	This command draws a line.	SPACE$(n)	This command inserts the number of spaces that are specified in the parentheses ( ).
PLAY	This command plays specific notes.	SPC(n)	This command is used to set a print point that is (n) number of spaces from the left of the screen.
PRINT	This command sends information to the screen.	STEP	This command is used with the FOR NEXT command to have the FOR NEXT loop to skip by the number stated.

Command	Definition	Command	Definition
RANDOMIZE	This command sets the star of the RND random number generator to simulate real random numbers.	SUB	This command starts the subroutine.
REM	Remarks or statements that are not included in the code, but are used for those observing the code later.	UCASE$(variable$)	This command changes the string variable$ to all uppercase letters.
RND	This command creates a simulated random number.	USING	This command is used with the PRINT to format the output of the variable to fit the format that is expected.

Variable Types

Now that you have worked with variables, here is a chart of the specific variable types used in QBasic. Each programming language has its own notation and definitions. These are included here to aid you in developing QBasic.

Data Type	Minimum Value	Maximum Value	Sample Declaration
String	0 characters	32,767 characters	DIM Words$ or DIM Words AS STRING
Integer	-32,768	32,767	DIM Number% or DIM Numbers AS INTEGER
Long Integer	-2,147,483,648	2,147,483,647	DIM Bignumber& or DIM Bignumber AS LONG
Single Precision	-3.0402823E+38	3.0402823E38	DIM Single! or DIM Single AS SINGLE
Double Precision	1,79769313486231D E -308	1,79769313486231D E +308	DIM precisecurve# or DIM precisecurve AS DOUBLE

Example QBasic variable types:

- String Variable: Name$ = “Laura White” or Name AS STRING = “Laura White”
- Integer Variable: Age% = 23 or AGE AS INT = 23
- Long Integer Variable: Salary& = 200000 or Salary AS LONG = 200000
- Single Precision Variable: Hourlywage! = 12.45 or Hourlywage AS SINGLE = 12.45

Debugging

Here are a few common methods for finding problems in your QBasic code.

- **Print variables at key points**: Before using variables throughout a program, create simple PRINT statements during the program to check the values of the key variables. If the variables are wrong, they will continue to be wrong throughout the program.
- **Manual line review**: Pretend you are the computer, and follow the program exactly as it is written This can be extremely time-consuming if you have a very large program.

- **Step through the program**: Use the debugger included with the QBasic editor by having the editor show you each step as it occurs in the program. To run stepping in QBasic:
    - Click **DEBUG**.
    - Click **STEP**.
    - Press **<F8>** to advance each line through the code.
    - Press **<F10>** if you want to skip subroutines or functions.
- **Use Breakpoints**: Instead of stepping through the entire program, use breakpoints to step through just a specific portion of the code. To run breakpoints in QBasic:
    - Move to where you want to start, then Press **<F9>**. QBasic will highlight the entire line.
    - Run the program (**<SHIFT>** + **<F5>**).
    - Press **<F8>** to advance each line through the code.
    - Press **<F10>** if you want to skip subroutines or functions.

Error Codes in QBasic

When you encounter an error in a very large program, it becomes difficult to find the specific issue or problem. In this condition, the computer usually is able to tell you the error that has occurred. The ERR variable stores the error code returned by the computer. In advanced programming, the program allows for potential errors and writes error handling subroutines. The following chart lists QBasic error codes that are stored in the ERR variable and their meanings:

Value	Error	Value	Error
1	NEXT without FOR	37	Argument count mismatch
2	Syntax error	38	Array not defined
3	RETURN without GOSUB	40	Variable required
4	Out of data	50	Field overflow
5	Illegal function call	51	Internal error
6	Overflow	52	Bad file name or number
7	Out of memory	53	File not found
8	Label not defined	54	Bad file mode
9	Subscript out of range	55	File already open
10	Duplicate definition	56	Field statement active
11	Division by zero	57	Device I/O error
12	Illegal in direct mode	58	File already exists
13	Type mistmatch	59	Bad record length
14	Out of string space	61	Disk full
16	String formula too complex	62	Input past end of file
17	Cannot continue	63	Bad record number
18	Function not defined	64	Bad file name
19	No RESUME	67	Too many files
20	RESUME without error	68	Device unavailable
24	Device time out	69	Communication -buffer overflow
25	Device fault	70	Permission denied
26	FOR without NEXT	71	Disk not ready
27	Out of paper	72	Disk-media error
29	WHILE without WEND	73	Feature unavailable
30	WEND without WHILE	74	Rename across disks
33	Duplicate label	75	Path/File access error
35	Subprogram not defined	76	Path not found

APPENDIX 2 - MAJOR CAUSES OF ERRORS

As we discussed in the introduction, errors are not caused by computer mistakes. Computer programming “bugs” or errors are usually attributed to three possible causes: syntax errors, run time errors, and logic errors.

Syntax Errors

If your program contains a syntax error, your implementation of the commands and notation does not meet the precise requirements of the programming language.

The syntax for the Phake programming language is as follows:

```
1:  Command - Variable - Linker - Variable
```

Therefore, the program line should read:

```
1:  MIX brown sugar With spoon
```

If the programmer writes:

```
1:  Mix With brown sugar spoon
```

Then the line violates syntax because it follows the form:

```
1:  Command - Linker - Variable - Variable
```

The QBasic editor does not allow you to have syntax errors. The editor reads each line of code when you press return and evaluates the syntax; however, not all programming languages have this capability, and they will let you make many mistakes before you discover the problem.

Run Time Errors

A run time error occurs when you have input that is not expected by the program. For example:

```
1:  PRINT “what is your mood today” : INPUT mood$
2:  IF mood$ = “happy” THEN PRINT “Great, have fun
    today”
3:  IF mood$ = “sad” THEN PRINT “Sorry to hear that,
    cheer up”
4:  END
```

If the user enters “fine,” the program will end with no output.

The QBasic editor cannot prevent run time errors, but the programmer can build error handling into the program by anticipating the errors and coding solutions for them.

QBasic has a line that can be added to the code called “ON ERROR Goto” that sends the program to an error handling subroutine. See “Appendix 1 - Additional QBasic Resources” on page 66 for the types of run time errors that are found by the special ERR variable in QBasic.

Logic Errors

Logic errors occur when you give instructions to the computer and it executes the code exactly as you programmed, but the results are garbled. This can be frustrating when you believe the logic is sound, but the computer operates contrary to your expectations.

The following example depicts a subroutine that manipulates a variable, which is also in the main body of the program:

```
1:  a = 5                ‘ sets a to 5
2:  b = 6                ‘ sets b to 6
3:  call sub oops (b)    ‘ goes to the subroutine
                           below and returns
4:  c = a + b            ‘ sets the c variable to the
                           product of a+b
5:  Print c              ‘ prints the c variable


1:  sub oops (b)
2:  b = 3                ‘ sets b to 3
3:  end sub
```

If you were not aware of the call to the subroutine oops, you would expect c to be equal to 11 (because it is the product of [a =5] + [b = 6]). It is not completely obvious that the subroutine oops changes the value of b to 3, making c equal to 8 (the product of [a = 5] + [b = 3]), and it may be unintentional due to duplicate variable names.

QBasic lets you make logic errors and, in fact, will help you make those errors. QBasic and all programming languages will follow all instructions given to them with no deviations. See “Appendix 1 - Additional QBasic Resources” on page 66 for more information on fixing logic errors in QBasic.

APPENDIX 3 - ASCII CHARACTER SET

ASCII, pronounced "ask-key," is an acronym for American Standard Code for Information Interchange. Computers can only understand numbers, so an ASCII code is the numerical representation of a character, such as “a” or “@,” or an action of some sort. ASCII was established to achieve compatibility between various types of data processing equipment.

The standard ASCII character set consists of 128 decimal numbers ranging from zero to 127, assigned to letters, numbers, punctuation marks, and the most common special characters. The Extended ASCII Character Set also consists of 128 decimal numbers and ranges from 128 to 255 representing additional special, mathematical, graphic, and foreign characters. Here is the standard ASCII chart with the decimal number, the hexidecimal number, and the ASCII character representation:

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

APPENDIX 4 - VOCABULARY EXERCISE ANSWERS

Match the vocabulary items on the top with their correct definitions on the bottom.

Compiler	**Debugger**	**Loop**	**Binary**	**Variable**
Syntax	**String Variable**	**Function**	**Constant**	**Flow Chart**
Subroutine	**Array**	**Phake**	**GUI**	**Literal**

___Array___	1.	A variable that holds a list of items that share the same properties or need to be grouped together.	_Function__	9. Used when you have only one variable that you need to pass between the main program and the sub program.
__Variable__	2.	A container that holds information.	_Debugger__	10. A tool to aid the programmer in removing all of the possible errors that can occur in computer programming.
__GUI_____	3.	Used to describe the interface where the actions that drive the computer are all based upon the user's input through visual navigation.	_Subroutine_	11. Used when you have more than one variable that you need to pass between the main program and the sub program.
__Literal___	4.	The name for the value that is contained within a variable.	_Compiler__	12. A programming tool that converts source code written in a high level language into machine code.
__Phake___	5.	A pseudo-code programming language invented for this class to illustrate programming logic.	_Loop_____	13. A continuous logic construct that evaluates a logic statement; if correct it continues the program, if not correct it continues to cycle.
Flow Chart	6.	A modeling tool used to best simulate computer logic or step-by-step instructions. It is very useful when there are multiple ways to produce an outcome.	String Variable	14. A type of variable that holds character based information.
_Syntax___	7.	The exact order or logical structure of a programming command. This word is used to denote the order in which a language is grammatically correct.	_Binary____	15. A numbering system that computers use to communicate and consists of just two unique digits, "1s" and "0s."
_Constant__	8.	A variable that holds a specific piece of information throughout the program.		

APPENDIX 5 - QBASIC EXERCISE ANSWERS

There are many ways to solve each programming challenge; these are possible solutions to the QBasic Exercises.

> **Exercise 1:** Change the operator in **NUMBERS.BAS** to addition. Be sure to change the user instructions so they reflect the functional differences of the program.

Solution:

```
1:  REM This works with simple numbers
2:  CLS
3:  PRINT "Enter two numbers that you would like to
    add"
4:  PRINT
5:  PRINT "Enter your first number": INPUT x
6:  PRINT "Enter your second number": INPUT y
7:  z = x + y
8:  PRINT "The first number plus the second number is";
    z
9:  END
```

Solution Notes:

- Line 3 changes from multiply to add.
- Line 7 changes the operator from * (multiply) to + (add).
- Line 8 changes from multiplied by to plus.

> **Exercise 2:** Modify **STRING1.BAS** so that the users also must enter their middle initial.

Solution:

```
1:  ' This is a sample string variable handling program
2:  '
3:  '
4:  CLS
5:  PRINT "What is your first name": INPUT first$
6:  PRINT "What is your middle name": INPUT middle$
7:  PRINT "What is your last name": INPUT last$
8:  name$ = first$ + SPACE$(1) + middle$ + SPACE$(1) +
    last$
9:  PRINT "Your name is "; name$
10: END
```

Solution Notes:

- Line 6 is inserted to ask the user for his or her middle name, thereby declaring a middle$ variable.
- Line 8 (was 7) now contains the middle$ and an additional space to the variable name$.

Exercise 3: Modify **STRING2.BAS** to correctly produce your Micron username.

Solution:

```
1:  ' This is a another string variable handling pro-
    gram
2:  '
3:  '
4:  CLS
5:  PRINT "What is your first name": INPUT first$
6:  PRINT "What is your middle name": INPUT middle$
7:  PRINT "What is your last name": INPUT last$
8:  name$ = first$ + SPACE$(1) + middle$ + SPACE$(1) +
    last$
9:  Upname$ = UCASE$(name$)
10: Lname$ = LCASE$(name$)
11: micron$ = LCASE$(LEFT$(first$, 1) + LEFT$(middle$,
    0) + LEFT$(last$, 10))
12: PRINT
13: PRINT "Your name is "; name$
14: PRINT
15: PRINT "Your name is "; Upname$; " in upper case."
16: PRINT
17: PRINT "Your name is "; Lname$; " in lower case."
18: PRINT
19: PRINT "Your Micron username should be "; micron$; "
    "
20: END
```

Solution Notes:

- Each solution is different depending on your individual Micron username.
- Line 11 now limits first$ to a single letter and specifies that the middle name contributes no letters to the username.

Exercise 4: Add a feature in **STEAK.BAS** so the program accepts cooklevel$ entries with any capitalization combination. For example, the program would accept RARE, Rare, or rare as correct entries. **Hint:** you only need to enter one line of code!

Solution:

```
1:  '  Cooking a Steak
```

```
2:  '  IF THEN ELSE example
3:  '
4:  CLS
5:  DIM cooklevel$, cooktime
6:  PRINT "How do you want your steak cooked?"
7:  PRINT "Please choose from rare,medium,medium well,
    and well done": INPUT cooklevel$
8:  cooklevel$ = LCASE$(cooklevel$)
9:          IF cooklevel$ = "rare" THEN
10:         cooktime = 2
11:         ELSEIF cooklevel$ = "medium" THEN
12:         cooktime = 5
13:         ELSEIF cooklevel$ = "medium well" THEN
14:         cooktime = 9
15:         ELSEIF cooklevel$ = "well done" THEN
16:         cooktime = 15
17:         ELSE cooktime = 20
18:         END IF
19: PRINT "Cook the steak for"; cooktime; " minutes on
    each side for a perfect steak"
20: END
```

Solution Notes:

- Line 8 is added to convert the cooklevel$ entry to lowercase letters.

Exercise 5: Alter the program **FORNEXT.BAS** so that the user inputs the repeat variable number. To save on time, you should also limit the entry to a value of 20 or below.

Solution:

```
1:  ' MY FORNEXT example for Programming Fundamentals
2:  '
3:  '
4:  CLS
5:  DIM repeat, looptimes, timesleft
6:  PRINT "Enter the number of times you would like
    this program repeated"
7:  PRINT "WARNING: Enter a number lower than 20!":
    INPUT repeat
8:          FOR looptimes = 1 TO repeat
9:                  SLEEP 1
10:                 PRINT "This will repeat itself ";
    repeat; " times"
11:                 PRINT
12:                 timesleft = repeat - looptimes
13:                 PRINT " Only "; timesleft; " more
    times left"
```

```
14:                 PRINT
15:         NEXT looptimes
16: END
```

Solution Notes:

- Line 7 is added and requires that the user input the value for repeat variable.

Exercise 6: Comment out line 6 of **DOWHILE1.BAS** and run the program to observe the effects. What happens?

.Solution:

```
1:  'Do while1
2:  CLS
3:  DIM pause
4:          DO WHILE (INKEY$ <> CHR$(33))
5:                  PRINT "This will print until you
    press the ! key";
6:  '                       FOR pause = 1 TO 10000: NEXT
    pause
7:          LOOP
8:  PRINT
9:  PRINT "You escaped!!"
10: END
```

Solution Notes:

- If you comment out line 6, the program scrolls so quickly that you cannot observe the program.

Exercise 7: Comment out line 11 of **DOWHILE2.BAS** to observe the effects. What happens?

Solution:

```
1:  'Do while2
2:  CLS
3:  DIM location, pause
4:          DO WHILE (INKEY$ <> CHR$(33))
5:                  location = location + 1
6:                          IF location = 40 THEN
7:                          location = 0
8:                          END IF
9:                  PRINT
10:                 PRINT SPC(location); "This will
    print until you press the ! key";
11: '                       FOR pause = 1 TO 10000: NEXT
    pause
12:         LOOP
13: PRINT
```

```
14: PRINT
15: PRINT "You escaped!!"
16: END
```

Solution Notes:

- If you comment out line 11, the program display changes from a waterfall effect to one that is difficult to observe.

Exercise 8: Modify **ARRAY1.BAS** so the user also enters the day after tomorrow (2 days from today). Be sure to change the number of items in the array and modify the printed text accordingly.

Solution:

```
1:  'My Array1 program
2:  '
3:  '
4:  CLS
5:  DIM dayofweek$(4)
6:  PRINT "What day of the week was yesterday": INPUT
    dayofweek$(1)
7:  PRINT "What day of the week is today": INPUT
    dayofweek$(2)
8:  PRINT "What day of the week is tomorrow": INPUT
    dayofweek$(3)
9:  PRINT "What day of the week is two days from now":
    INPUT dayofweek$(4)
10: PRINT
11: PRINT "THANK YOU!"
12: PRINT
13: PRINT "You entered: "; dayofweek$(1); ", ";
    dayofweek$(2); ", "; dayofweek$(3); ", and two days
    from now is "; dayofweek$(4)
14: END
```

Solution Notes:

- The dayofweek$ array in line 5 now contains 4 items.
- Line 9 is added to include the input.
- Line 13 improves the statement to include dayofweek$(4).

Exercise 9: In **ARRAY2.BAS**, if the user indicates that today is either Saturday or Sunday, the program will not function correctly because 0 (1 - 1) and 8 (7 + 1) are not defined in the array. Improve the logic of the program so the program will run correctly.

Solution

```
1:  'My Array2 program
2:  '
```

```
3:  '
4:  CLS
5:  DIM dayofweek$(7), Inputday
6:  dayofweek$(1) = "Sunday"
7:  dayofweek$(2) = "Monday"
8:  dayofweek$(3) = "Tuesday"
9:  dayofweek$(4) = "Wednesday"
10: dayofweek$(5) = "Thursday"
11: dayofweek$(6) = "Friday"
12: dayofweek$(7) = "Saturday"
13: PRINT "Enter the number that corresponds to the day
    of the week"
14: PRINT
15: PRINT "Sunday is        1"
16: PRINT "Monday is        2"
17: PRINT "Tuesday is       3"
18: PRINT "Wednesday is     4"
19: PRINT "Thursday is      5"
20: PRINT "Friday is        6"
21: PRINT "Saturday is      7"
22: PRINT
23: INPUT Inputday
24: IF Inputday = 1 THEN PRINT "Yesterday was Saturday,
    Today is "; dayofweek$(Inputday); ", and Tomorrow
    is "; dayofweek$(Inputday + 1): END
25: IF Inputday = 7 THEN PRINT "Yesterday was ";
    dayofweek$(Inputday - 1); ", Today is ";
    dayofweek$(Inputday); ", and Tomorrow is Sunday":
    END
26: PRINT
27: PRINT "THANK YOU!"
28: PRINT
29: PRINT "Yesterday was "; dayofweek$(Inputday - 1);
    ", Today is "; dayofweek$(Inputday); ", and
    Tomorrow is "; dayofweek$(Inputday + 1)
30: END
```

Solution Notes:

- Line 24 is added so that if the value of 1 is entered, then the program has an alternate ending.
- Line 25 is added so that if the value of 7 is entered, then the program has an alternate ending.

Exercise 10: Add a new question in the main program of **SUBROUTE.BAS** that asks the user to input a description of the person whose name was entered. Then pass that variable information to the subroutine for concatenation.

Solution:

```
1:  DECLARE SUB combo (name$, description$, message$)
2:  'A simple example of subroutine
3:  '
4:  '
5:  CLS
6:  PRINT "This is a sample of how subroutines work"
7:  PRINT
8:  PRINT
9:  PRINT "Please enter your name": INPUT name$
10: PRINT "Please enter a description of this person":
    INPUT description$
11: CALL combo(name$, description$, message$)
12: PRINT message$
13: END


1:  SUB combo (name$, description$, message$)
2:  message$ = name$ + " is " + description$
3:  END SUB
```

Solution Notes:

- Line 1 now contains a new variable description$.
- Line 10 is added and prompts the user to enter a description.
- Line 11 now contains a new variable description$.
- Line 1 of SUB combo now contains a new variable description$.
- Line 2 of SUB combo altered to concatenate the name with the description.
- Notice that the IF THEN statement and extra code is deleted in the SUB combo.

APPENDIX 6 - REFERENCE GUIDE

WHAT'S NEXT?

Class Resources

The following books are beneficial for those who need more information about the basics of programming:

- *Absolute Beginner's Guide to Programming*, Greg Perry
- *Beginning Programming for Dummies*, Wally Wang
- *SAMS Teach Yourself Beginning Programming in 24 Hours*, Greg Perry
- *Introduction to the Personal Software Process*, Watts S. Humphrey

Books24x7

Books24x7 is an external online IT reference library that all Micron team members can use. Books24x7 offers hundreds of books about programming, from very basic to advanced.

To register to use Books24x7:

1. From the MERC, click **DEPARTMENTS > MORE**.
2. Click **LIBRARY**.
3. Click **BOOKS24X7** on the top menu bar.
4. Click **REGISTER TO USE BOOKS24X7.COM**.

To log on to Books24x7:

1. Go to www.books24x7.com
2. Click **FIND BOOKS**.
3. In the Topic List, click **PROGRAMMING**.

Courses at Micron

- **MC4081** Intro to Perl Programming
- **MC6405** HTML Web Page Basics
- **MC6407** HTML Forms and Cold Fusion
- **MC9031** Visio Basics
- Plus numerous programming courses available from XtremeLearning

XtremeLearning

XtremeLearning offers hundreds of self-paced training courses about desktop, IT, and business and professional courseware. XtremeLearning courses can be taken from any Internet connection at work or home at any time. The following programming courses are recommended to expand your programming knowledge:

- Internet & WWW Introduction
- MS Visual Basic 6.0
- CIW Perl Fundamentals
- C++ for Non Programmers
- C Programming
- C++ Programming

If you already have an existing XtremeLearning account, then you can simply modify your training plan to include any of the above-mentioned courses. If you do not have an XtremeLearning account, you will need to register with the site and enroll in courses.

To log on to XtremeLearning for the first time:

1. Go to www.xtremelearning.com from any Internet connection at work or home.
2. Enter the *Self Registration ID*: ***comicron***
3. Enter the *Password*: ***elearning2001***
4. Click **NEXT**.
5. Enter your personal information. Be sure to use your Micron username as *Username* and write down your password. You can use your Micron password or create a new one.

   Note: This password will not be updated with AMS because it is separate from Micron's internal environment.

6. Click **NEXT**.
7. Verify your information, and then click **DONE**.
8. After you have logged into XtremeLearning for the first time, click **FIRST TIME USERS** and complete the tutorial.

For more information about XtremeLearning at Micron, visit the internal XtremeLearning web site at http://hercules.micron.com/is/support/trnres/J3/xtreme.htm.

BIBLIOGRAPHY

1. Bradley, Julia Case & Millspaugh, Anita C. *Programming in Visual Basic 6.0*. New York, NY: McGraw-Hill/Irwin. 1999.
2. Kahane, Howard & Tidman, Paul. *Logic & Philosophy: A Modern Introduction*, 7th Edition. Belmont, CA: Wadsworthy Publishing Company. 1995.
3. Perry, Greg. *Absolute Beginner's Guide to Programming*, 2nd Edition. USA: Que. 2001.
4. Perry, Greg. *SAMS Teach Yourself Beginning Programming in 24 Hours*. USA: Sams. 1998.
5. Potter, Richard E., Rainer Jr., R. Kelly, & Turban, Efraim. *Introduction to Information Technology.* New York, NY: John Wiley & Sons, Inc. 2001.
6. Wang, Wallace. *Beginning Programming for Dummies*. Foster City, CA: IDG Books Worldwide, Inc. 1999.
7. Zak, Diane. *Programming in Visual Basic 6.0*. Course Technology, Inc. 1999.
8. Books24X7.com, Inc. 1999-2001. <www.books24x7.com>