



# *Computer Concepts and Applications* *Spring 2018*

*A Textbook for CSCI 130 & 150*  
*College of St. Benedict/St. John's University*

*Noreen Herzfeld and John Miller*  
*with contributions from*  
*Becca Simon, Josh Trutwin, Andrew Whitford Holey & Lynn Ziegler*



# Table of Contents

*Page*

## *Introduction .....v*

Preface to the Student: Read This To See What You're Getting Into!

## *Chapter 1: What Computers Do .....1[1]*

- 1.1 General vs. Special Purpose Computers
- 1.2 What Computers Do
- 1.3 Programs and Algorithms
- 1.4 The History of Computing
- 1.5 Conclusion

## *Chapter 2: Applications Programming ..... 2[1]*

- 2.1 Higher Level Programming Languages
- 2.2 Translation Programs
- 2.3 Algorithms and Program Design
- 2.4 Steps in Writing a Program
- 2.5 Object-Oriented Programming
- 2.6 Conclusion

## *Chapter 3: An Introduction to Visual Basic 2013.....3[1]*

- 3.1 Forms and Projects
- 3.2 Starting Visual Basic and Creating Objects
- 3.3 Declarations, Assignment Statements, and Order of Precedence
- 3.4 Writing Code for a Simple VB Project and Saving a VB Project
- 3.5 Input and Output
- 3.6 Numeric and String Functions
- 3.7 Conditional Statements
- 3.8 Loops
- 3.9 Using Data Files
- 3.10 Arrays
- 3.11 Searching
- 3.12 Sorting
- 3.13 Using Multiple Forms
- 3.14 Writing and Using Your Own VB Functions
- 3.15 Conclusion



# Table of Contents

*Page*

## **Chapter 4: Databases: An Introduction to Access 2016 .....4[1]**

- 4.1 Basic Database Principles and Features
- 4.2 Designing a Database
- 4.3 Creating Tables in Access 2016
- 4.4 Select Queries
- 4.5 Advanced Queries
- 4.6 Forms and Reports
- 4.7 Relationships between Tables and Relational Queries
- 4.8 Macros in Access
- 4.9 Connecting to an Access Database with a VB program
- 4.10 Conclusion

## **Chapter 5: Spreadsheets: An Introduction to Excel 2016 .....5[1]**

- 5.1 Introduction to and Uses of spreadsheets
- 5.2 Entering and Formatting Data
- 5.3 Cell References, Functions, and Formulas
- 5.4 Sorting and Conditional Functions
- 5.5 Graphs and Templates
- 5.6 Macros
- 5.7 Example VBA Macros
- 5.8 Conclusion

## **Chapter 6: How Computers Work: Data Representation .....6[1]**

- 6.1 Analog and Digital Signals
- 6.2 Overview of Computer Design
- 6.3 Binary Representation of Positive Integers
- 6.4 Rules of Binary Addition
- 6.5 Two's Complement Representation of Positive and Negative Integers
- 6.6 Floating Point Representation for Real Numbers
- 6.7 Representing Text and Symbols
- 6.8 Representing Pictures
- 6.9 Representing Sound
- 6.10 Conclusion

**Chapter 7: The CPU and Machine Language .....7[1]**

- 7.1 The CPU
- 7.2 The Fetch-Execute Cycle
- 7.3 Machine Language
- 7.4 A Simple 8-bit Computer
- 7.5 Assembly language
- 7.6 Conclusion

**Chapter 8: Computer Hardware, Operating Systems and Networks .....8[1]**

- 8.1 Overview of Computer Hardware
- 8.2 Input and Output
- 8.3 Auxiliary Storage
- 8.4 Systems Software
- 8.5 Operating Systems
- 8.6 Networks
- 8.7 Conclusion

**Chapter 9: Data Manipulation .....9[1]**

- 9.1 Data Manipulation: Boolean Operations and Circuit Components
- 9.2 Boolean Expressions, Circuit Diagrams, and Truth Tables
- 9.3 Simplifying Boolean Expressions
- 9.4 Designing Circuits
- 9.5 Conclusion

**Chapter 10: Social and Legal Issues .....10[1]**

- 10.1 Privacy
- 10.2 Security
- 10.3 Ownership and Copyright Laws
- 10.4 The Computerization of Society

**Appendices .....A[1]**

- [A: Visual Basic Reference Guide](#)
- [B: Excel VBA Macro Reference Guide](#)
- [C: Access VBA Macro Reference Guide](#)

## ***Introduction:***

### ***Preface to the Student: Read This to See What You're Getting into!***

This book is designed for students taking an introductory course in computing at St. John's University and the College of St. Benedict. This course has a dual function. It provides an introduction to computer programming as well as to the major application tools, spreadsheets and databases, while also satisfying the requirements for a natural science course in computer science. It is aimed at students who may take only one computer science course while in college, yet also serves as a first course in computing that can lead to a major or minor in computer science. No previous knowledge of or experience in using computers is assumed, other than the ability to use a word processor to type up lab reports. The goal of this course is to help you become a comfortable and competent user of standard applications, with enough knowledge to further your computer skills on your own while also providing a foundation for further study in computer science.

The material covered in this book falls into three sections. The first section of the book provides an introduction to the programming process. Computers do not function without software, or programs. Although few of you will become computer programmers, knowing how programs are designed and written will help you better use programs or direct others who do so. In a business or personal setting you will be better equipped to decide when to purchase programs and when to ask for custom designed programs and you will know what sorts of things are possible and impossible for a program to do.

To illustrate the basics of programming, this book uses Visual Basic 2013, a widely used programming language for both business and personal use. A second reason for introducing you to Visual Basic is that a version of Visual Basic, VBA, underlies the programs in Microsoft Office. Thus, knowledge of Visual Basic allows you to write your own macros to tailor spreadsheets and databases to your own particular needs.

The second section presents the two applications of the computer most commonly used in a business setting, spreadsheets and databases. The material in this section will equip you not only to use both Excel 2016 and Access 2016 in standard applications, but also to write simple programs and macros in VBA, allowing you to go beyond the functions built into these packages and to tailor applications to specific needs. While we will not cover all of the options available in these application programs, by the end of this book, you should have sufficient command of the vocabulary and understanding of the workings of these applications to be able to further your knowledge on your own. What you don't learn in this course you should be able to look up in a reference book or in on-line help pages as you need it.

The third section explains how computers store and manipulate data. The purpose of this section is to describe the workings of the computer in sufficient detail to demystify what goes on inside the machine. Knowledge of how computers function will help you understand why you can and cannot do certain things when you work with computers. It will also help to know the basic components of a computer system when you purchase and work with computers in the future, or when you communicate with those who provide you with computer support.

Finally, this book provides extensive reference sheets in the appendices, so that you can look up how to do things as you need them in laboratory exercises or in working on your own projects. Examples of programs, spreadsheets, and databases will be available on-line and your instructor will give you directions on how to access them.

A basic understanding of how computers work and how to use standard business applications is virtually a necessity in today's world. Acquiring this basis will increase your flexibility and give you the skills to better organize and analyze data, as well as meeting future job requirements. I also hope that you will find the process of gaining new skills and designing computer applications can be both rewarding and fun.



# *Chapter 1*

## *What Computers Do*

### *1.1 General vs. Special Purpose Computers*

You have all had experience using computers. You may have worked with computers in school or at work and even done some programming. You've probably surfed the Internet and played a computer game or two. You may have downloaded music or pictures. These are the applications we all immediately think of when we think of computers. The machines we use to accomplish these tasks range from a notebook or laptop computer to a desktop machine, to a much larger mainframe computer at a business or school. These machines generally have a variety of programs on them and can do different things depending on which program we choose to run. Thus, they are called general-purpose computers.

What you may not realize is the number of other computers that you work with on a day to day basis. There are computers embedded in virtually every appliance these days, from your car to your microwave to your DVD player. These are special purpose computers, designed and programmed to do one specific task. There are also computers running the infrastructure of our society. Computers are integral to banking, merchandising, transportation, and the financial and inventory apparatuses of most businesses. They also control the phone network and the light, water, and sewage grids. Thus, each of us comes in contact, both directly and indirectly, with many computers in the course of most days. Advances in the technology of silicon chips, which have made computers both smaller and less expensive, together with the development of sophisticated computer software have allowed computers the flexibility and power to become an integral part of our lives. Thus, although we first think of general purpose machines when we think about or hear of computers, it is important to remember that the majority of computers that are manufactured and with which we come into contact are special purpose computers.

### *1.2 What Computers Do*

In this book, however, we will focus on the operation and some major applications of general purpose computers. To begin, we will address the question of what goes on inside that box on your desk. How do computers do the many things they do? With so many different applications, such as mathematical computations, word processing, games, spreadsheets, surfing the net, storing music, etc., understanding how each of these is accomplished looks like a major task. However, it's not as complex as it looks. To understand how a computer accomplishes a task we can break the task down into its basic components and then see how each of those components is accomplished. As you might imagine, many computer applications share similar

components. In fact, a surprisingly small set of operations is sufficient for all the different things computers do. We can see what processes are necessary by looking at two examples: computing the answer to an equation and writing a paper with a word processing program.

When we ask the computer to find the answer to an equation such as  $A = 2 * 3^2 + 6/2 - 7$  the computer needs to be able to store numbers and execute a variety of mathematical functions: add, subtract, multiply, divide, raise to a power, take a square root, etc. For the more complicated functions such as taking a number to a power, or finding a square root, there are methods that use the standard arithmetic functions of adding, subtracting, multiplying, and dividing. These methods can be entered into the computer as programs that use only those arithmetic functions. For example, taking a number to a power can be reduced to multiplying that number by itself the proper number of times, i.e.  $4^3 = 4 * 4 * 4$ . Multiplication itself can be understood in terms of addition,  $3*4 = 3 + 3 + 3 + 3$ . Division can be programmed using the method of subtracting the divisor from the dividend until the divisor is greater than the dividend remaining, the answer consisting of the number of subtractions made. For example, to find  $6/2$  we see that  $6 - 2 = 4 - 2 = 2 - 2 = 0$  uses three subtractions before the dividend is less than the divisor, so  $6/2 = 3$ . Subtraction is the same as adding a negative number. Thus we can represent the equation above as:

$$\begin{aligned}
 A &= 2 * 3^2 + 6/2 - 7 \\
 &= 2 * (3 * 3) + 6/2 - 7 \\
 &= 2 * (3 + 3 + 3) + 6/2 - 7 \\
 &= (3 + 3 + 3) + (3 + 3 + 3) + 6/2 - 7 \\
 &= (3 + 3 + 3) + (3 + 3 + 3) + 3 + (-7)
 \end{aligned}$$

So all the processor really needs to know how to do is store both positive and negative numbers (and fractions) and add. Anything more can be accomplished through the use of programs. We may also want to check the computer's answer, asking it if  $A = B$  or to do something if  $A > 100$ . Thus the computer also needs a way to tell if two numbers are equivalent or if one is larger than another.

Now consider the sort of things we do when we use a word processor. We enter a long string of characters from the keyboard. This string includes letters and numbers, spaces, tabs, carriage returns, etc. The computer needs the ability to store each of these. To edit our document, we might move certain words or sentences or we might run it through a spell check. To check spelling the computer needs to compare one string of characters to another. Thus for word processing we need the ability to store text, move text, and make comparisons.

If we were to consider other applications such as games or working with a spreadsheet or database, we would see that they would reduce to a similar list of operations. All tasks done by a general purpose computer can be reduced to pretty much the same operations needed in the two scenarios above.

### **Basic computer processes:**

- 1) store all kinds of data, including numbers, text, pictures, and sound
- 2) move this data from one storage location to another
- 3) compare any two pieces of data to determine if they are equivalent or if one is larger (or later in the alphabet, or darker, or lower in pitch) than the other
- 4) add

We are going to take a top-down approach to exploring computing and look at higher level interactions with a computer first before coming back to study the lower level basic operations in Chapters 6 through 9. There you will learn about the inner workings of a computer's processor, showing how the computer stores, moves, compares and adds data and how data is input into and output from most general purpose computers and transmitted from one computer to another. These operations are all that need to be built into the actual machinery or hardware of the computer. This is not to say that most computers do not have a larger set of operations built in, but the four capabilities listed above, together with input and output are the minimum needed to have a functioning computer. Understanding how these operations happen will be enough to get a basic idea of how computers work.

### ***1.3 Programs and Algorithms***

Since no general purpose computer has all the operations one would wish for hard-wired into its circuitry, we also need some way for the computer to execute more complicated processes. This is accomplished through the use of software, programs that are stored in the computer and run as needed. Computer programs are formal representations of methods for performing computations. The programs themselves are written in a code or programming language that can be understood by the computer. Since computers can only do exactly what they are told to do, these programs must be both detailed and well organized. Every program is based on a method for computing the desired result called an algorithm. Every algorithm has the following characteristics:

#### **Definition of an algorithm:**

- 1) it consists of a list of steps that complete a task
- 2) each step is precisely defined and appropriate for the machine for which the algorithm is designed
- 3) the number of steps is finite
- 4) the process terminates after a finite number of steps are executed

While it may look like numbers 3 and 4 are the same, they aren't. The steps in an algorithm may be repeated, thus it is possible to have a finite number of steps that repeats one or more of them infinitely. An algorithm must come to an end at some point. Also, each step must be precise and appropriate for the machine the algorithm is designed for. For example, a step in an algorithm might be to turn 90 degrees to the right. This would be an appropriate step for a mobile robot, but would not be appropriate in an algorithm for your desktop computer. A step that just said "turn" would probably not be precise enough.

Suppose, for example, you wished to program a computer to add the numbers from one to ten and print the result. One possible algorithm for this process would be the following:

Example:

- 1) begin with  $\text{sum} = 0$  and  $x = 1$
- 2) add  $x$  to  $\text{sum}$
- 3) add 1 to  $x$
- 4) if  $x < 11$  go back to step 2
- 5) print the  $\text{sum}$
- 6) stop

There are six steps to this algorithm. Each step is precise and could be done by a person with a pencil and paper. Steps two through four are repeated, but once  $x > 10$  the process terminates with step six. Thus this algorithm meets the characteristics for a valid algorithm.

Not all algorithms are mathematical. A cooking recipe is an algorithm for a human being to carry out, as are the instructions for assembling a bicycle or operating a washing machine. Each of these is a finite list of precise steps, appropriate (though not always easy) for humans to carry out, that terminates at some point.

Once we have an algorithm for a process, the performance of that process does not take any insight, only the ability to follow directions. Thus, if an algorithm exists for a task and each step is mechanically possible (we would not ask a computer to blow its nose or climb a tree) we can program a computer to accomplish the task. If there is no algorithm for a task or if the algorithm requires capabilities not inherent in a given machine, then we cannot program that machine to perform that task. The design of good algorithms is central to computer science.

Computers are not intelligent. Any intelligence is built into the design of the algorithms that form the basis of their programming. Without programs, computers can do only a few very basic things. Yet computers are extremely powerful. Their power to do many things comes from two properties. First, computers are very fast. A complicated program may break a task into millions, or even billions of steps. However, a computer can execute each of those steps in nanoseconds, or billionths of a second. Computers can accomplish in seconds what it would take a human being hours, days, or even years to compute. Second, once we have programs to do simple things we can combine those programs to do more complicated things. From our set of operations -- storing, moving, comparing and adding data -- we can build an incredibly diverse variety of programs using different algorithms. This gives computers their great flexibility. Any process that uses data or information that can be symbolized and that can be described by an algorithm can be accomplished by a computer.

## ***1.4 The History of Computing***

Computing is a very old idea which has taken on new meaning with the advent of modern computers. The word *computing* comes from Latin and refers to the process of doing

arithmetical calculations. The related word *computer* originally meant a person who does computing. Mathematicians used the term computing to describe the study of methods for doing calculations. When machines were developed which could perform these methods, they were naturally referred to as computers, and it would probably be insulting to apply the term computer to a person today.

The first computing machine was probably the abacus, though we believe that Stonehenge and the pyramids of the Mayans to also have been built in order to do some sort of astronomical calculations. Computers based on gears were developed by Pascal, Leibnitz, and Charles Babbage in the seventeenth, eighteenth, and nineteenth centuries, respectively. Babbage's machine was the closest in design to today's computers, as it was programmable through the use of punched cards, a technology that had been developed in the early nineteenth century in order to set automatic looms to weave different patterns. A friend of Babbage, Lady Lovelace (Augusta Ada Byron), is considered the first computer programmer.

The first electronic computers were developed during World War II in order to crack enemy codes and to calculate trajectories for launching shells. These first machines used mechanical relays, a technology that was quickly replaced by vacuum tubes. The first vacuum tube computer was the ENIAC, developed by Eckert and Mauchly at the University of Pennsylvania. The ENIAC set the standard for computers through the nineteen-fifties. Vacuum tube computers are called the *first generation* of electronic computers. They were large machines, primarily programmed in machine code, prone to mechanical malfunctions, and extremely expensive and difficult to maintain.

The *second generation* of computers was introduced in the mid-sixties following the invention of transistor technology. While the standard vacuum tube had been four to five inches long, it could now be replaced by a 1/2" transistor. This change in scale made computers both smaller and faster. Since many more transistors could be connected in a small space, second generation computers had much larger memories than first generation. This change in memory size led to the development of more advanced programming languages and the application of the computer to a much wider range of problems. This, combined with the relative economy and reliability of transistors, led to much more widespread use of computers. Where before computers were used only by governments and large businesses, second generation computers were now within range of mid-sized businesses and schools.

The *third generation* was introduced in the late nineteen-sixties with the advent of integrated circuit technology. Rather than using separate tubes or transistors for each circuit, the circuits that make up the computer were now etched on a silicon chip. Whereas one transistor was roughly 1/2" by 1/2", now the entire circuitry of the computer could be etched onto a single chip of the same size. This change in size effected a corresponding change in computing speed. A computer that would have previously required a room to store could now sit upon a desk top. Computers became physically smaller while having much greater memory capacity. They have also become much cheaper and easier to maintain, bringing computation within the reach of almost all businesses and into most homes. The machines we use today are still based on this

integrated circuit technology, though the circuits themselves have become smaller and faster over the last twenty years.

Where will computer technology go in the next twenty years? Every major development in computer hardware has come about by making the components both smaller and faster and there are several avenues scientists are exploring to continue this process. One is optical computing, in which data will be transferred via light rather than electricity. Since light photons have no mass, light can be transmitted without either heat or friction, which allows light to move significantly faster than the electrons in a silicon based system, which do have mass. New advances in fiber optics make optical systems a viable possibility in the near future. Other approaches to computing in the future include nanotechnology, which is exploring the possibility of making computers as small as a molecule, with their encoding stored in single atoms, and quantum computing, which also hopes to develop atom sized computers that work according to the theories of quantum mechanics. A third approach is to base computers on DNA, using the nucleotides of DNA to code and store information. While these are all in the preliminary stage of development, it seems clear that the computers of the mid- twenty-first century will be based on a hardware quite different from those of today. On the other hand, throughout all these changes in hardware, the basic concepts on which computing is based have not changed. Since this course is being taught from a top-down approach, we will first learn about solving problems with the computer by writing programs in a high level language and using existing applications and then we will come back to gain a better understanding of the underlying basic concepts in the second half of the course.

## ***1.5 Conclusion***

Computers have become a major part of the world around us. Most computers are special purpose computers, embedded in other appliances and machinery and designed to accomplish a single task. However, when we think about computers, most of us think of general purpose computers, those designed to process a variety of programs and types of data. Although these computers seem to accomplish an endless array of tasks, each task can be broken into smaller components. Although computers calculate, process words, search and sort, play games, and store and transfer data great distances, at the hardware level they only need the built-in abilities to store and move data, compare, and add.

Programs allow the computer to go beyond these basic operations. They provide the power and flexibility of the computer by combining basic operations into more complex ones. Each program is based on an algorithm, a finite, terminating list of steps to accomplish a task. Before one can write a program for a computer to do some task, one must design an algorithm for that task. The upcoming chapters will explore the process of learning to use the computer to assist you in solving problems. This will entail using existing software (applications), designing your own solution to a problem (algorithms), coding your solution (programming), and combining all of these steps into creating a solution to a problem of your choice (projects.)

## *Chapter 2*

# *Applications Programming*

### *2.1 Higher Level Programming Languages*

To make programming an easier task it would be nice to have computers that understood human languages, such as English. This is beyond the capabilities of today's computers for many reasons. Most human languages use a very large vocabulary, many words and phrases have multiple meanings, and meaning is often derived from context, rather than from the words themselves. This makes the task of understanding human language too difficult to program into a computer. What we need is to meet the computer half way. Higher level programming languages do just that.

When a high level programming language is used, the programmer can more easily focus on the details of solving the problem and not have to think about what kind of computer is being used. Higher level programming languages allow the programmer to think on a more abstract level and not to have to pay attention to the details of how the data is moved between memory and the CPU. Since these details are determined only later, in the process of translation, most higher level languages are machine independent. This means that programs written in a higher level language are portable; they can be used on a variety of different machines. Most higher level programming languages use English phrases for commands and standard mathematical notation for equations and they assume a much larger set of operations than might be built into any given CPU.

Like the operating system, higher level programming languages provide another layer of abstraction between the user and the actual machine. They allow the user to ignore much more of what goes on at the machine level and to think in terms of much larger constructs. The user can focus on the problem to be solved rather than on the details of the machine being used.

### *2.2 Translation Programs*

In order for a computer to execute or run a program, it must be translated into the lowest level language, machine language, which consists solely of codes written using only ones and zeroes. Since the computer can execute only machine language programs, it is necessary to have programs that translate higher level code into machine language. This is a difficult task since there is no one-to-one correspondence between higher level language statements and any given computer's machine language. A single line of a high level language program may translate into

many lines of machine language code. For example, a higher level language program might include a statement such as:

$$\text{IF } A > B \text{ THEN } A = (A + 1) - 2$$

As you will see later when we study machine language later in the semester, this statement would translate into eight lines of machine language. The point we want to make here is that it is much easier to write code to solve problems in a higher level language, but when a higher level language is used then translation is necessary and a single line of high level code may represent many lines of the lower level language the computer requires to run the program.

There are two approaches to translating a higher level language. Before we discuss them, let's look at the analogy of translating from one human language to another. There are two approaches to this sort of translation as well. At the United Nations, interpreters translate a delegate's speech into multiple languages simultaneous to the delivery of the speech. As soon as the speaker completes a sentence the interpreter translates that sentence into another language. This doesn't always result in the smoothest translation but it allows for the translation to occur with no delay. Translating one sentence at a time, however, would not be a very good approach to translating a poem from one language to another. While one might get the gist of the poem's meaning, the result would not be very poetic and probably would not carry any of the nuances of the original. To translate a poem one must consider the poem as a whole, decide what it is conveying to the reader and then construct a similar poem in the new language.

The two types of translating programs for higher level languages work in much the same way as the above examples. Translating programs take a program from the original high level language, called the *source code*, and translate it into machine language, called the *object code*. An *interpreter* is a program that translates one line of source code at a time, like the interpreter at the UN, while a *compiler* is a program that takes the source code as a whole and constructs an equivalent program in the object code.

Interpretation happens simultaneously with the execution of a program. The interpreter takes the source code one line or code segment at a time and constructs equivalent object code, then sends this object code to the CPU for execution before translating the next line or code segment. Interpretation is good for programs that are highly interactive with the user, but has several drawbacks. Interpreted programs slow down execution, since the computer must stop after each chunk of code and wait for the next segment to be translated. Also, since translation and execution are simultaneous, the program must be retranslated every time it is executed.

Compilers avoid these problems by translating the entire program before execution. The compiler goes through the entire source code and constructs an equivalent program in the object code. Thus after compilation there are two versions of a program stored in the computer, the original higher level language version and an equivalent machine language version. Only the machine language version is executable. The advantage of compilation is that if a program is run many times the translation process need only happen once. Once a machine language version of the program is produced and stored that version can be run any number of times. Another advantage is that most compilers are able to optimize code; in other words, the compiler finds



ways to make the code run faster or take up less space in memory. Thus, after the initial compilation, compiled programs are much more efficient than interpreted programs. However, compilers are extremely complex programs and may take up a great deal of memory space themselves.

## 2.3 *Algorithms and Program Design*

A program is only as good as the algorithm that underlies it. When writing a program in a higher level language it is imperative to first design a clear algorithm. To see the steps of the algorithm clearly, we generally write it out first in a notation called *pseudocode*. Pseudocode represents a compromise between English and the programming language we plan to use. While it uses the basic structure of a programming language, it omits many of the details.

One may need several tries before coming up with the most efficient algorithm for a task. Let's consider the example of finding the smallest number in a list of numbers. Start by asking yourself how you would solve the problem without using a computer. If the list were short you'd probably see the smallest number at a glance. This doesn't help much in designing an algorithm. But if the list were long, you'd probably look at a smaller portion of the list, find the smallest number in that portion, then continue with the next portion, seeing if it had a number smaller than the one you already found. Here we have a method we can develop into an algorithm.

1. find the smallest number in the first portion, call it S1
2. find the smallest number in the next portion, call it S2
3. compare S1 to S2 and let S1 equal the smallest of S2 and S1
4. repeat steps 2 and 3 until all the numbers have been checked
5. S1 contains the smallest number in the list

Now we need to refine our algorithm to make it more specific. For example, what is a portion of our list? How long should a portion be and how do we find the smallest number in this portion? The shorter the portion, the easier it will be to find the smallest number in it. So let's consider the shortest possible portion, that of size 1. Finding the smallest number in this portion is trivial. Our algorithm would now be as follows:

1. let the first number in the list be S1
2. compare the next number, S2, to S1
3. let S1 equal the smallest of S2 and S1
4. repeat steps 2 and 3 till all the numbers have been checked
5. S1 contains the smallest number in the list

To write this algorithm in pseudocode, we would probably add some mathematical notation to our steps. One possible pseudocode version would be:

1. S1 = first number in list
2. Do until end of list  
    S2 = next number in list  
    If  $S2 < S1$  then  $S1 = S2$
3. Print S1

The final step would be to test this algorithm on a short list. Suppose we try the list 45, 13, 62, 11, 12. Let's make a chart of the values S1 and S2 take on as we work through the algorithm. Such a chart is called a *trace* and is a good way to check both algorithms and programs for errors in the logic or errors in the code. (errors in the code are known as “bugs” and tracing is a way of “de-bugging”)

<u>S1</u>	<u>S2</u>	<u>Output</u>
45	13	
13		end of first time through steps 1 and 2
	62	step 2
	11	
11		end of third time through step 2
	12	fourth time through step 2, end of list reached
		11 the program will print the smallest value. Hooray!

There will still be things to make more specific when we write the final program. For example, how will the computer know when the end of the list has been reached? How will it retrieve the numbers in the list? These are things that depend on the specifications of the programming language, so we will leave them to be specified later. The main thing is that we now have a specific method for solving our task and we know it works.

## 2.4 Steps in Writing a Program

Writing computer programs is time consuming and requires a level of attention to detail that many people are unused to. The process is easier and the result more likely to be error free and to deliver the results desired if one follows the following steps.

### 0. Decide Whether to Write or Buy

This step is labeled step 0 because it is not really part of writing a program, but is a step you might wish to consider before writing a program. There's no point in reinventing the wheel. If you can find a commercial program that does pretty much what you want, it is often cheaper to buy than to build your own.

### 1. Analyze the Problem -- Output and Input

Producing a computer program has something in common with the dreaded word problems of mathematics. Usually the problem is presented in words, i.e. write a program to calculate the bill owed by a customer or write a program to help a farmer determine how many acres of soybeans to plant this year. The first thing you need to figure out is exactly what the task is that the computer needs to perform. A good way to approach this question is to determine what output you wish the program to produce. Once you know what you want as your output, determine what information you need as input in order to get to this output. Give the inputs and outputs names that can be used in your program and introduce any other names for values needed to assist you

in solving the problem. The names for data values are called variables. Earlier in section 2.3 the variables S1 and S2 were introduced to represent numbers in a list.

## **2. Design the Algorithm**

The next step is to sketch out the steps that will be needed to get from the inputs to the output. At this point you do not need to worry about any particular programming language. Write out the steps of your algorithm in English or in pseudocode. Introduce any variables needed to solve the problem and also make your algorithm easy to follow and understand. Run through a simple test case with a pencil and paper to make sure your algorithm does what you want it to. Your algorithm will typically involve using, manipulating, and/or calculating values of the variables that you have defined to represent the problem at hand.

## **3. Design the User Interface**

Determine how you wish the screen to look for the user of this program. Make sure you have made adequate provision for the user to enter inputs and to view any output. Make your interface as clear and user friendly as possible, including instructions for the user whenever needed.

## **4. Write the Code**

The next step is to translate your algorithm into the programming language you plan to use. If you have written out the algorithm in sufficient detail, this step becomes quite simple; you need only determine what code you need for each step.

## **5. Test and Debug the Program**

The next step is to run the program to see if it actually does what you intended. Be sure to test your program on a variety of inputs. Sometimes the program will work fine for some, but not for others. Most programs do not work perfectly the first time. Errors in a program are called bugs. There are a variety of processes for locating the bugs in your code and fixing them.

## **6. Document the Program**

The final step is to add notes to your program describing what the program does and how it works. While this step is not absolutely needed in order to have a functional program, it is very important. Often, programs are used in a business setting for years and may need to be modified long after the original designer is gone or has forgotten how the program was designed. Good documentation allows a new programmer to quickly see how a program works and to understand what its various components do.

We construct software, and the process is not that different from constructing anything else. Suppose you wished to construct a scooter for your cousin. The first thing you might do is check to see if you could buy a scooter for less or perhaps find a do-it-yourself kit that would work well. If not the next step would be planning; you need to determine what kind of scooter you

want to build and what materials you wish to use. Then you would draw up a plan for the scooter and come up with an algorithm for its construction consisting of the steps you will go through to build it. The actual construction of the scooter according to your plans is analogous to the programming stage. Before you give the scooter to your cousin, you would want to take it for a test drive to make sure it works properly and is safe. You might find that you need to make a few changes. Finally, you might wish to keep your plans, in case your younger brother wants a scooter next summer, and you might wish to write down some instructions for your cousin. This would be analogous to the documentation phase.

## 2.5 *Object-Oriented Programming*

There are several approaches to writing computer programs. One of the approaches is known as the *imperative programming paradigm* where the instructions that make up the program are in a list and the computer executes these instructions from beginning to end. The structure of this type of program has one instruction always following the previous one in a predetermined order.

A different approach to designing a program is the *object-oriented* paradigm. To understand this paradigm, think of a computer screen with several icons on it. Each of those icons represents a different task or computer file, each with its own attributes and procedures. A user can click on these icons in any order, depending on what that user wants the computer to do.

Object-oriented programs work in much the same way. The program itself is a collection of objects. Each object has certain *properties*. Some objects also have code associated with them that makes something happen, called a *method*. The different objects in a program interact with each other as needed through these methods. A method is executed through an *event*, such as a mouse click, a keystroke, or opening a form. Objects that share certain properties or methods are grouped together in a *class*.

If you think for a moment about an object-oriented environment, it is not so different from the real world. The world is populated with objects, each object having certain properties. Some objects form natural classes, such as the set of all tables, chairs, or cars. Each object in one of these classes shares certain properties with all other objects in the class, though each object might also have properties it does not share. Objects interact with each other in certain ways. Consider, for example, my car. It has properties that all objects in the car class share, such as wheels and an engine. It has other properties that are not shared by all cars, such as the color red, manual transmission, and an age of 12 years. There are methods associated with this car, such as running the car by the event of starting the motor, or rolling down the driver's window. These methods and events change properties of the car; if I start the motor the property “running” might change from no to yes. Opening the window changes the property “windows closed” from yes to no. Painting the car might change its color property from red to blue.

Other events or methods might involve other objects. The method “hit squirrel” might not change any properties on my car but it changes the squirrel's property of living from yes to no.

The method “head on collision” would change many properties, not only of my car but of the other car as well!

Object-oriented programs thus have their primary focus on the objects that are manipulated by the program. The methods that change these objects can be made to occur in a variety of orders, giving object-oriented programs a level of flexibility not found in imperative programs. Objects can also be shared among a variety of software systems, making program development easier, since one can store away libraries of objects, each with their own properties and associated methods.

Writing an object-oriented application calls for a few small changes to the list of steps for writing a program in section 2.4.

### **1. Analyze the Problem -- Output and Input**

The programmer still needs to begin by deciding exactly what the task is that the computer needs to perform and determining the desired output or outcome. The outcome in the case of an object-oriented program will be a change to some object. For example, the object changed might be a textbox on a screen, the change being that an answer gets printed in it.

### **2. Design the Objects and Methods**

Before designing an algorithm, the object-oriented programmer needs to decide what objects will be needed and may want to specify some initial properties for those objects. A list of objects and their important properties should be generated. The next step is then to determine how those objects will interact through methods. Each method will need its own algorithm, a description of the steps that make up that method. Thus in object-oriented programming, instead of designing one long algorithm, the programmer will generally design several separate algorithms, one for each method. As before, these algorithms should be written out in pseudocode and tested.

### **3. Design the User Interface**

Since the user interface is generally an object, much of this has probably been done in step 2. However, this is a good time to give the interface some extra attention, making it as pleasant to look at and as user friendly as possible.

### **4. Write the Code**

The next step is to translate all the event algorithms into code.

### **5. Test and Debug the Program**

Each event needs to be tested and debugged separately. After this, you will also want to try triggering the various methods in different orders, to make sure they all work in the desired manner, no matter which events have already occurred.

## 6. Document the Program

This is just as necessary for object-oriented programs as for imperative ones. Documentation should include a list of all objects with their associated properties and events, descriptions of each event, and a discussion of how the different objects can interact with each other.

## 2.6 *Conclusion*

Although the computer needs machine language programs to operate, machine language is not easy for human beings to write and is at such a basic level that programs are both lengthy and complicated. Higher-level languages allow programmers to think in larger conceptual units, to use English words and regular mathematical notation, and to ignore all the particulars of the machine's architecture. Since the machine can only execute machine language programs, higher-level programs must be translated into machine language, either during execution with an interpreter or before execution with a compiler.

Although it is fairly easy just to sit down at a terminal and write a short program once you are familiar with a programming language, this is not advisable. There are distinct steps that a programmer should follow when writing a new program. For programs that have more than a dozen lines of code, following these steps will result in both a better product and in the avoidance of a lot of frustration. Get used to writing out your plans and algorithms for a program before writing any code. You will thank yourself for it later.

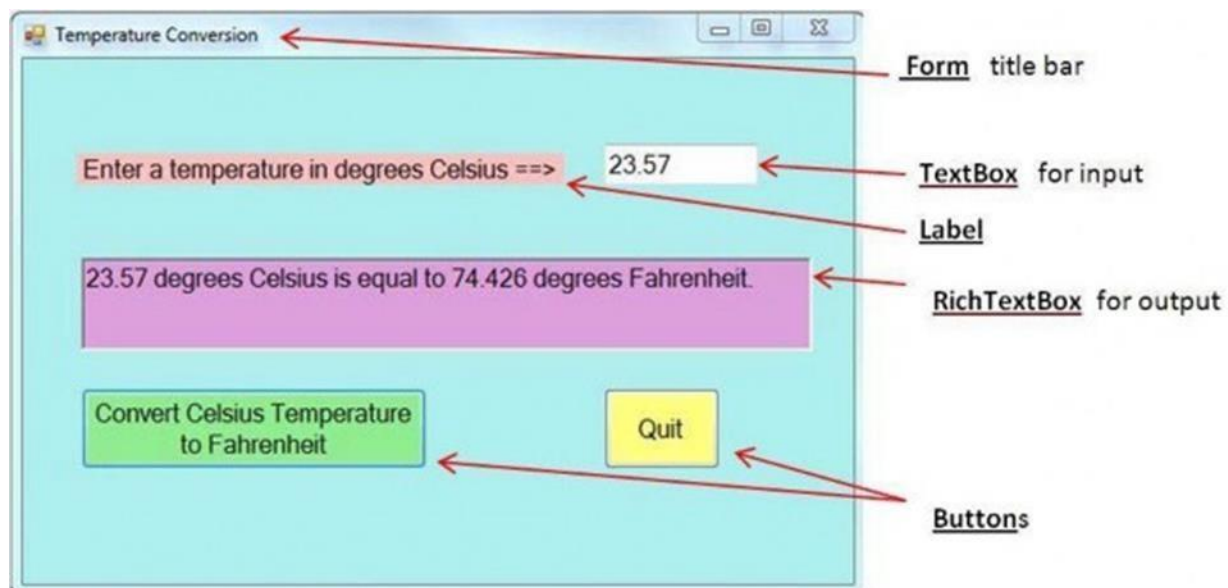
## Chapter 3

### *An Introduction to Visual Basic 2013*

Basic is an imperative programming language developed in the nineteen-sixties as a general purpose language that would be easy to learn and use. Basic was widely used in schools, homes, and small businesses throughout the 70s and 80s, but began to decline in popularity in the late 80s as more highly structured languages became the norm. Visual Basic (VB) is a newer, object-oriented version developed by Microsoft so that users of Windows could write their own applications programs. Visual Basic is also the language that underlies all the Microsoft Office programs, such as Excel and Access. Thus Visual Basic provides a tool that allows a user to link multiple Microsoft applications. Knowing Visual Basic not only allows you to write your own programs, but it also gives you the ability to tailor Excel spreadsheets or Access database applications to your own needs.

#### **3.1 Forms and Projects**

As you might guess from the name, Visual Basic is visually oriented. In other words, a visual basic program has a *graphical user interface* (GUI) through which the program gets its input and displays its output. One of the first things a VB programmer must do is design this interface. If the application is complex, there might be several different screens that a user will see at different times during the application. These screens are called *forms*. A typical form might contain text, pictures, boxes for input, boxes to show output, or buttons to cause some event to happen. A VB form is displayed below.



The form itself, the boxes for input and output, the labels, and the buttons are all objects. Each one has certain properties, such as size, color, text. While most of the objects in a VB program appear on a form, some objects might be separate from a form, such as a database or a spreadsheet. Some objects, especially the buttons, will have events associated with them. Recall that an event makes something happen to change the properties of an object. With the form on the previous page, the “Convert Celsius Temperature to Fahrenheit” button calculates and displays the result in the RichTextBox used for output from this program. The quit button ends the program and makes the entire form disappear from the screen. Each event will have a program associated with it, the list of instructions the computer needs to make the event happen. The instructions associated with a form’s events are saved when you save your form and appear in a code editing window when you want to edit them. Generally, each form in a VB project will have an associated segment of VB code that includes code for each event that can occur from that form. In this course, typically the code will be the programs you write for the “click event” for each button on your form. The VB code for the form on the previous page is listed below. Note there are two sections of code that begin with “Private Sub” and end with “End Sub”. Those are the two small programs or subroutines associated with each of the two buttons on the form that handle the tasks to be accomplished when each button is clicked, (a click-event). Don’t expect yourself to understand the code at this point. It will be explained later in the chapter.

```
Option Explicit On
Public Class frmTemperatureConversion
    'This program accepts a Celsius temperature value input by the user
    ' and then calculates and displays the corresponding Fahrenheit temperature.

    'Sample algorithm to accomplish this task:
    '1. get a Celsius temperature from the user, call it celsiusTemperature
    '2. Convert celsiusTemperature to the Fahrenheit scale and call the result fahrenheitTemperature
    ' (Use the formula:  $F = (9/5)C + 32$ )
    '3. Display the results in a complete statement including both temperatures.

    Private Sub btnConvert_Click(sender As System.Object, e As System.EventArgs) Handles btnConvert.Click
        'declare the variables
        Dim celsiusTemperature As Single = 0.0, fahrenheitTemperature As Single = 0.0

        'assign the user-input from the textbox to a variable
        celsiusTemperature = txtCelsius.Text

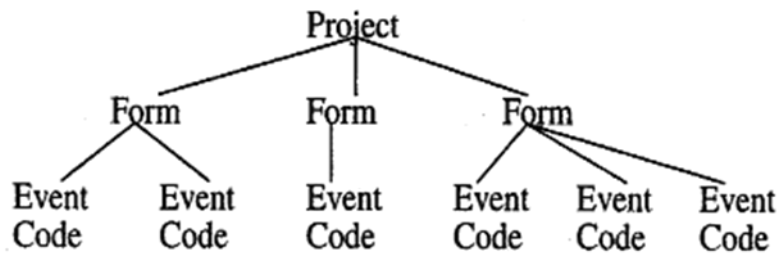
        'perform the calculation and assign the result to a variable
        fahrenheitTemperature = 9 / 5 * celsiusTemperature + 32

        'display the result in a textbox on the form
        outResults.Text = celsiusTemperature & " degrees Celsius is equal to " &
            FormatNumber(fahrenheitTemperature, 3) & " degrees Fahrenheit."
    End Sub

    Private Sub btnQuit_Click(sender As System.Object, e As System.EventArgs) Handles btnQuit.Click
        'stop the run of the program
    End Sub
End Class
```



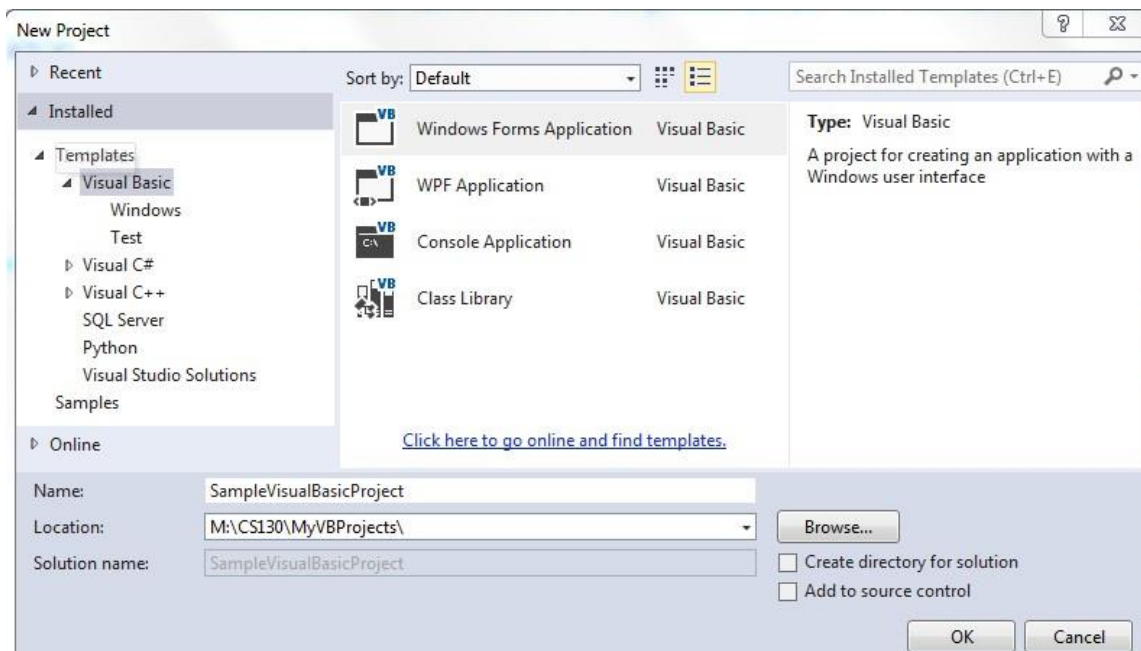
A diagram indicating the organization of a Visual Basic project is listed below:



The programmer first must decide how the overall project should be designed. The first step of this is determining how many forms are needed. Next, each form should be designed with the objects needed for input, output, and events. The next step is to design, write, and test algorithms for each event. Finally, the code for each event is written, tested, and debugged.

### 3.2 Starting Visual Basic and Creating Objects

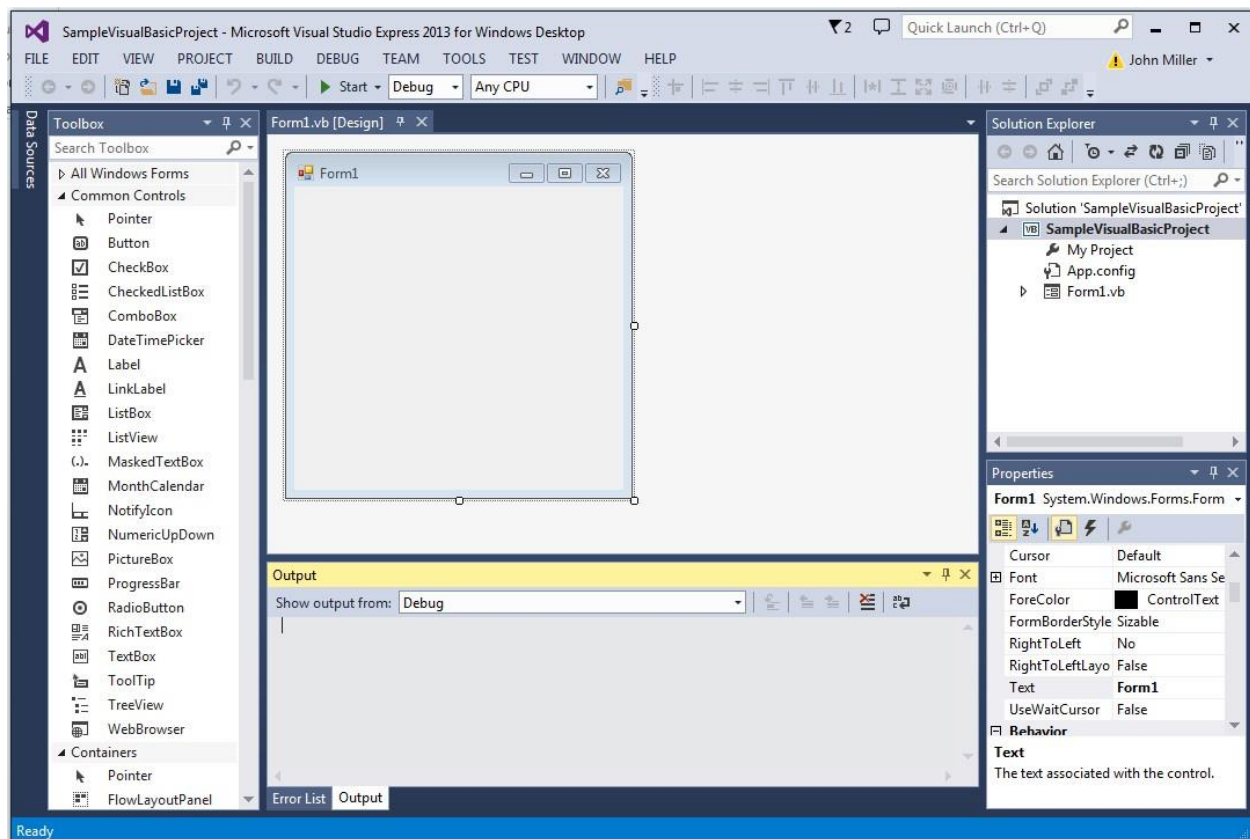
The Visual Basic 2013 programming environment on our campus computers is installed as part of a larger program, Visual Studio Express 2013 for Desktop. You can find the program, “**VS Express 2013 for Desktop**”, on CSB/SJU computers from the Start Menu under “All apps”, or use the search window near the start icon. When you start Visual Studio for the very first time you may need to choose the programming environment you wish to work in. Visual Studio is a platform that can be used with many different programming languages. In this course we will be programming in Visual Basic 2013. To create a new VB project, choose “New Project” from the menu. You will see a screen like the one below.



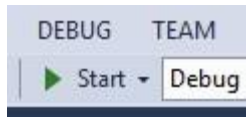
Make sure that Visual Basic has been highlighted under the Templates tab to indicate the language you will use and Windows Forms Application is chosen as the type of project. Those choices may be the default, but if they are not, you can choose the language and type of project to create. Once you have made your initial choice to work in Visual Basic (VB), the next time you start Visual Studio, it will launch right into the VB environment.

Give your project a name right away in the name pane at the bottom of the screen and also browse to the location, (folder), you wish to save your project. The “Solution name” pane will be automatically filled in with the name of the project. Make sure that the box next to “Create directory for solution” is unchecked.

Click **OK** to begin a new project. This will bring up a window with a new, blank form and several other panels with information about the project as you can see in the next figure.

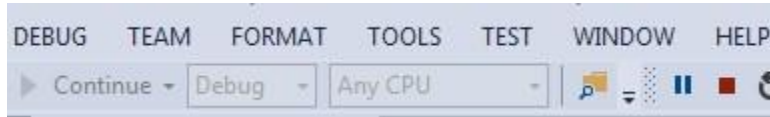


At the top of the window is the menu bar and just below it is the toolbar. Using choices from either the menu or the toolbar will allow you to do a variety of things to your project or form. The tool bar below has icons for the most common functions, such as beginning a new project, adding a new item to your project such as a form, saving your project, and running your project. The icon you will probably use the most is the small green triangle near the middle of the toolbar with “Start” next to it, shown below, which you click on to run your programs and check for errors.



(click on the green triangle to run your program)

After you press the green triangular run icon, a small square icon to the right of the triangle will turn red, see below.

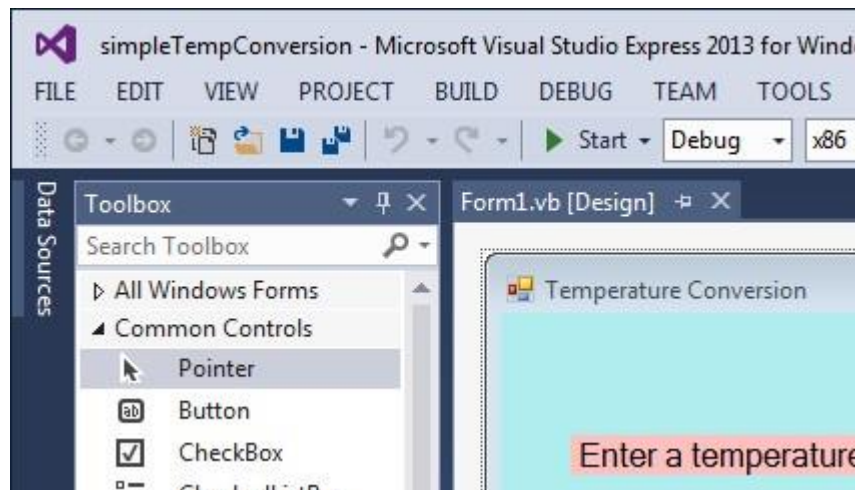


Pressing the red square icon is one of the ways you can stop your program and return to design mode. You **must** stop your program before attempting to make changes in it.

On the upper left side of the screen is a blank form labeled “*Form1*” that you will use to create and display the graphical user interface for your program. You can resize the form by dragging one of the corners or sides. The Solution Explorer panel in the upper right displays the structure of your project. You can move to a different form, if a project has more than one, by double-clicking on the desired form.

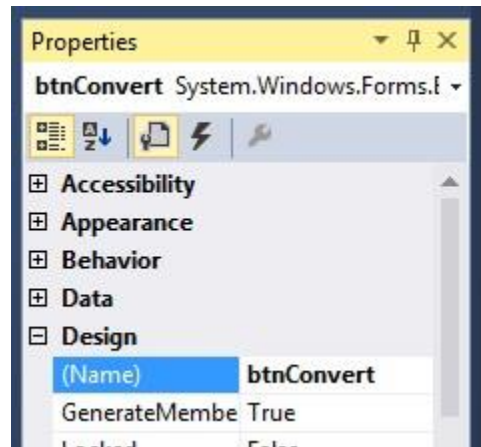
If the toolbox tab, the properties window, or the Solution Explorer panes are not visible, you can use the “View” menu to activate any or all of those to be visible.

The Toolbox tab is on the upper left side of the Visual Studio window by default. You will use the Toolbox to select objects to place on the form(s) as you create the graphical user interface to your program. If you do not want the toolbox to be visible all of the time, you can unpin it by clicking the stick pin icon which will change to AutoHide mode. The default setting for the Toolbox tab is to be Docked or “pinned”, making it visible.



You will most often use the *Common Controls* tab within the toolbox that shows a list of all the different objects that can be added to a form. To add an object to a form, simply click on the icon once and then move your mouse over the location on your form where you would like the object

to be placed and draw the boundaries of the object with the mouse. The object that is currently being added or modified will have a border of squares around to show that it has been selected and those small white squares can be used as handles to change the size. To select an object for modification, click on it. On the bottom of the right side of the Visual Studio window is the *Properties* panel. A portion of it is shown below:



This window shows all the possible properties for the current object. VB gives each object a default name when it is first created, such as Form1 or Button1. While you could use the default names, they are easy to confuse with each other when you have multiple objects of the same type on a form (i.e. Button1, Button2, Button3). For this reason, it is important to rename each object as you place it on a form. To avoid confusing object names with variable names in the program, a standard way of naming objects has been developed that will be used throughout this course. The basic idea of the naming scheme is to put a prefix in front of the object name that tells what type of object it is. The object will then be referred to by this name in the code for any event that uses this object. See the “Object Naming-Conventions Table” to see the prefix for the various objects you will be using in your coding and the minimal list of properties for each that should be changed when they are created. Additional properties may also be changed as desired. Note that Object names should **not** contain spaces.

**Object Naming-Conventions Table**

Object	Prefix	Name	Object properties to change
Form	frm	frmFormName	Name Text
TextBox	txt	txtTextBoxName	Name
Label	lbl	lblLabelName	Name Text
RichTextBox	out	outRichTextBoxName	Name Font
Button	btn	btnButtonName	Name Text

Use names that describe the function and type of object (i.e. btnComputeTotal, btnQuit, etc.)

## Objects

The most important objects for a simple VB application are the form, textboxes, rich textboxes, labels, and command buttons, (as shown in the diagram in section 3.1). We will look at the main properties of each of these objects.

### *Form*

The form provides the background for the user interface. You can think of it as a blank canvas on which to build your application. When setting up a new form, the main properties that you might wish to change are the *Name* and *Text*. Set *Name* to something that will remind you of this form's purpose. (Remember to use the prefix, *frm*, when naming the form.) Changing *Text* changes the text that appears on the title bar of the form. You may also wish to change other properties such as the *BackColor* property. To change the *BackColor*, click on the *BackColor* property row, then the down arrow button that appears, then the Custom tab button on the top, and then the color desired. You could also resize the form by clicking on it and dragging the small boxes on the sides, if desired. You can reposition where the form will appear on the screen during program execution by changing the *StartPosition* property of the form.

### *TextBox*

A textbox, as its name implies, is a box that holds text. This text can be entered at design time or at run time, making text boxes an easy way for a user to enter input into a program while running it. The only property of a textbox that needs to be set on creation is *Name*. The *Text* property is blank by default. Any text entered at run time will become the new value of the *Text* property.

### *Label*

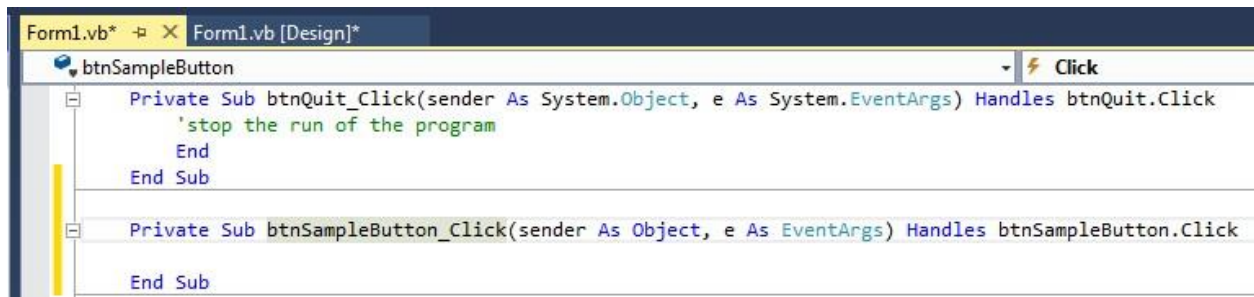
When using textboxes for input, it is important to give the user instructions on what to enter in the box. Thus each textbox should have a label next to it. Like a textbox, a label is simply a box that holds text, but labels are usually set at design time and not altered during program execution. If you set the *BackColor* of the label to the same color as the form only the text will be visible.

### *Button*

Buttons are the easiest way to trigger events from an application. While the program is running, when a user clicks on a button, an associated program will be executed that initiates some action or change on the form. The properties you will want to set when adding a button to your form are the *Name* (this is very important as this *Name* will be used in the associated code), and the *Text* on the button. To change the color of a button you can change the *BackColor* property to the color you want.

To add code so that a button performs an action, double click on it. This will bring up a code editor in a window where you can write the VB code associated with that button (and see all of the code that has already been written for that form.) Note that the Private Sub and End Sub lines are automatically added for you the first time you double-click the button. Following is an image showing the editor window after double-clicking a button with no code associated with it. You can also see in this image the code associated with the Quit button. You would write your code

for this button between Private Sub and End Sub. Click the [Design] tab at the top of the code window to return to the form designer.



### *RichTextBox*

Rich text boxes are the objects we will use to hold output generated by the application that we would like to have displayed on screen. Remember to name the rich text box using the “out” prefix. To use a rich text box for output, simply change the *Name* property at design time and change the *Font* property to Courier New. (This is important to ensure that all output is formatted correctly. All characters in Courier New are the same width, allowing us better control in aligning columns of output. We will discuss how to form columns later.) You will leave the box empty at design time so that it is available for output to be written in during the program’s execution using the *AppendText* method or by setting the *Text* property, as discussed later.

## **3.3 Declarations, Assignment Statements, Variable Scope, and Order of Precedence**

### *Declarations*

When writing a computer program, the programmer must decide what data is needed to solve the problem at hand. The names given to the data are called variables. The programmer must also know what type of data those variables are meant to represent. The program needs to know if the variable is meant to be interpreted as a numeric value, (ie. celsiusTemperature), or as a string of literal characters, (ie. firstName). Visual Basic uses the **Dim** statement to allow the programmer to declare the names of variables, set the data type for the variables, and give the variables an initial value. The idea of data type is very important so that the program uses the variables in the way intended by the programmer. All variables should be declared using one or more Dim statements and given an initial value. Often the value of the variable will change when the program is run, but giving it a known initial value is useful and can help reduce errors in your code. Initializing the variable when it is declared also reinforces the important concept of data type. A variable declaration has the form:

Dim *variableName* As *data type* = initial value

Following are several example Dim statements. Note the last example shows how you can declare several variables with a single Dim statement by separating them with a comma:



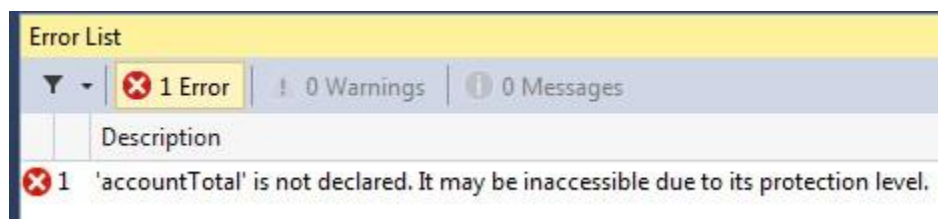
```
Dim celsiusTemperature As Single = 0.0
Dim age As Integer = 21
Dim firstName As String = "Hal", hourlyPay As Double = 0.0
```

The variable name is simply a letter or string of characters beginning with a letter. Variable names **cannot** contain spaces. It is good programming practice to use *meaningful variable names* with letters or words that correspond to the function of the variable (like `celsiusTemperature` for the Celsius temperature and `fahrenheitTemperature` for the Fahrenheit temperature.) It is common programming practice to begin variable names with a lowercase letter. All subsequent words in the variable name begin with a capital letter, such as `dateOfHire`. This is referred to as camelback notation. The data type determines how the information in the variable will be represented and stored inside of the computer. We will discuss data types in more detail later in the course.

Here is a list of some of the most common data types and the ones you will most likely use in this course.

Data type	Sample	Description
Integer	239	Small-to-medium-sized positive and negative whole numbers
Long	6,543,123,876	really-big-to-really-small positive and negative integers
Single	37.8992	Small-to-medium-sized positive and negative real numbers
Double	5.47E+68	really-big-to-really-small positive and negative real numbers
String	45 Third Ave.	a sequence of letters and/or characters as in a name or address
Boolean	True	Used to represent a true or false value True or False are the only values allowed
Date	7/4/1776	Used to represent dates or date and time- enclose in # symbols (i.e. Dim myBirthday As Date = #2/14/1995#)

If you forget to declare a variable, the Visual Basic software will display an error when you attempt to run your program, letting you know that you have not defined that variable. Here is a sample error message when the variable “`accountTotal`” was used without declaring it:



Some words are not available as variable names. Words that are already used by the system to name objects or their properties (such as Image, Picture, Height, Label) or words that have a function in VB code (such as End, While, Print) are called *reserved words* and cannot be used as variable (or object) names. You will receive an error message if you try to use a reserved word for something other than its predetermined use.

### *Assignment Statements*

Assignment statements have the form:

variableName = expression

Note that the variable is always on the left side of the equal sign in an assignment statement. Once a variable has been declared and initialized using a Dim statement, you can assign a new value to the variable at any time using an assignment statement. Variables can be declared only once, but can be assigned multiple times. Below is a code sample showing variable declarations and initializations followed by several assignment statements that change the initial values given to those variables. The expression on the right side of an assignment statement can simply be a value, (ie. myTemperature = 98.6) or a more complex expression where calculation is required before the assignment is completed, (ie. myPay = hours \* rate). The computer will perform the needed calculation automatically and then assign the value of the expression to the variable on the left side of the assignment statement. A detailed description of an assignment statement follows.

```
Dim salesTaxRate As Single = 0.07, celsiusTemperature As Single = 0.0
Dim age As Integer = 9
Dim bookTitle As String = ""

celsiusTemperature = txtCelsius.Text
bookTitle = "The Road Less Traveled"
age = 23
```

The Dim statements declare the data type of every variable and that information is used by the program. In the code above, celsiusTemperature is a variable of data type Single. Therefore, the assignment statement, celsiusTemperature = txtCelsius.Text, will automatically convert the text from the textbox into a Single (decimal number) value before assigning it to the variable. The order in an assignment statement is very important. The value of the expression on the right-hand side of the “=” symbol is assigned to the variable on the left-hand side. (variable = expression) If the expression on the right is also a variable, note that the variable on the left is the one that is assigned a new value.

### *Variable Scope*

The location within your program where you declare a variable determines its *scope*, or what code within your program has access to that variable. A variable declared within a subroutine or function, (sections of code bounded by Private Sub and End Sub or Function and End Function),



can only be used by that subroutine or function and is referred to as a “local” variable. A variable declared at the top of the form, not within any subroutine, is accessible to all the events for that form. A variable declared at the top of a form is referred to as a “form-level” variable. Variables should be declared locally unless there is need to have greater access to them. Restricting access to variables can help prevent accidental changes in the values of variables and makes the code easier to debug. To debug a program means to eliminate the errors, (get rid of the bugs.)

### *Order of Precedence*

Arithmetic operations and expressions use the following symbols:

Operation or expression	Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/
Integer division	\
Integer remainder	Mod
Exponentiation	^
Is greater than	>
Is greater than or equal to	>=
Is less than	<
Is less than or equal to	<=
Is equal to	=
Is not equal to	<>
Parentheses	( )

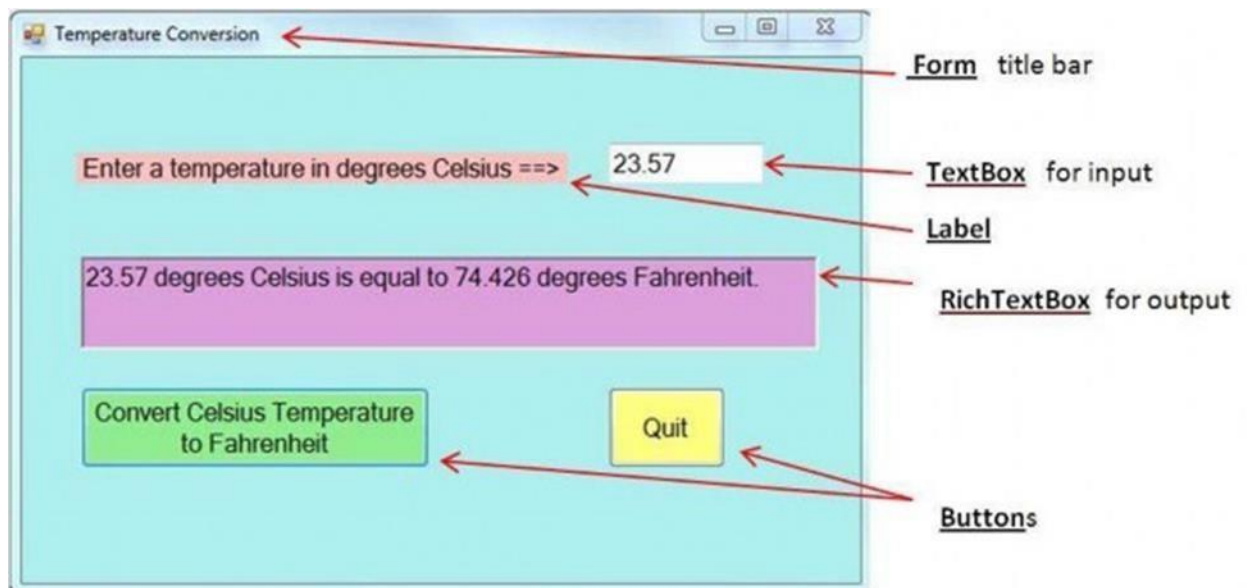
When more than one operation appears in an expression, the order these operations are executed in is called the order of precedence. The order of precedence for arithmetic operations in VB is Parens ( ) then powers ^ then mult and division \* / then addition and subtraction + -

Here is an example:

$$\begin{aligned} A &= 80 - (3 * 2)^2 * 2 + 7 \\ &= 80 - \underline{6^2} * 2 + 7 \\ &= 80 - \underline{36} * 2 + 7 \\ &= \underline{80 - 72} + 7 \\ &= 8 + 7 \\ &= 15 \end{aligned}$$

The computer would first multiply 3 times 2, getting 6, and then 6 raised to the 2nd power, getting 36. It would then multiply by 2, getting 72, and then subtract 72 from 80, getting 8 and finally adding 7, assigning the value 15 to A.

### 3.4 Writing Code for a Simple VB Project and Saving a VB Project



Let us consider the code needed for the form above. Double clicking on a button brings up a code module where the VB code can be written and edited. For example, when you double click on the convert button, the button name appears on the code module as follows:

```
Private Sub btnConvert_Click(sender As System.Object, e As System.EventArgs) Handles btnConvert.Click
End Sub
```

These lines mean that you are going to write a subroutine (Sub) or a short program that will be executed when the user clicks on the command button named "btnConvert". The click is an "event" and your code determines what actions will take place when that "event" of clicking the btnConvert command button happens. The subroutine ends with the words End Sub. Private

means that this subroutine can only be accessed from this form. The programmer now adds the code needed between these two lines.

The code to convert from Celsius to Fahrenheit would be based on the following algorithm:

1. get the Celsius temperature from the user (via the textbox)
2. convert the Celsius temperature to Fahrenheit using the formula:  
$$F = 9/5(C) + 32$$
3. display the Fahrenheit temperature in the rich text box

The code for this subroutine listed below introduces you to another very important part of good programming practice, adding *comments* to your program. Comment lines begin with an apostrophe, ' , to distinguish them from lines of code. Comments are for the benefit of the reader of the program; anything following the ' is ignored by the computer. Comments help a user or programmer know what the program is doing. Comments should be interspersed throughout the program, describing the action at various points.

The code for the temperature conversion subroutine would be:

```
'this is a program to convert Celsius temperatures to Fahrenheit
'declare the variables
Dim celsiusTemperature As Single = 0.0, fahrenheitTemperature As Single = 0.0

'assign the user-input from the textbox to a variable
celsiusTemperature = txtCelsius.Text

'perform the calculation and assign the result to a variable
fahrenheitTemperature = 9 / 5 * celsiusTemperature + 32

'clear the rich text box before displaying result
outResults.Clear()

'display the result in a RichTextbox on the form
outResults.Text = celsiusTemperature & " degrees Celsius is equal to " & vbNewLine
outResults.AppendText(fahrenheitTemperature & " degrees Fahrenheit.")
```

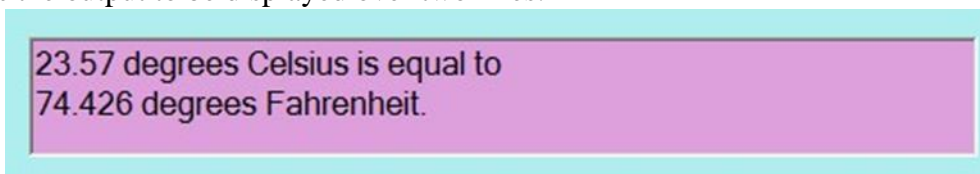
The first line of real code, the Dim statement, after the two comment lines is used to declare the variables. The next line of code, `celsiusTemperature = txtCelsius.Text` , gets the input from the textbox named `txtCelsius` and assigns that value to the variable, `celsiusTemperature`. Since the input is the Text in this box, we ask, not for the box itself, but for its Text (`txtCelsius.Text`). `celsiusTemperature` is a variable that represents this decimal number value. Note that in using an assignment statement the variable being assigned the value is always on the left. The next line does the calculation, again using an assignment statement, putting the result in the variable `fahrenheitTemperature`. The final two lines display the result in the rich text box named `outResults`. The line that begins with “`outResults.Text =`” is used to assign part of the result to the rich text box and the next line uses the `AppendText` method to add more text to what was already put there by the previous line. (`AppendText` adds text to whatever is already in the rich text box while assigning something new to the `Text` property of the rich text box replaces what

was there.) Whenever a variable such as `celsiusTemperature` or `fahrenheitTemperature` is to be printed, the computer will print the value in that variable. When text appears between quotation marks the computer prints exactly what is between the quotes. The ampersand (&) is needed as a concatenation operator or “joiner” that joins words and variables together into a longer string of characters. The ampersand is often used in creating the output to be displayed on the screen, as it usually involves combining words with the value of variables into meaningful sentences.

The VB language has some built-in variables. `vbNewLine` represents the new-line character and causes the display to move to the next line at the place it is appended within the String. The `vbNewLine` at the end of the line of that starts with `outResults.Text=` ,

```
outResults.Text = celsiusTemperature & " degrees Celsius is equal to " & vbNewLine
```

causes the output to be displayed over two lines:



Using `vbNewline` makes it easier to create multiple line output formatted as you wish. Note the use of `outResults.Clear()` in the code. This is an optional statement. Without the `Clear()`, if the user clicked on the compute button once, and then a second time, the results for the second calculation would appear in the line below the first. To clear the rich text box of any previous output so that only one output at a time appeared, you would need to use the `Clear()` method before displaying the new results. `Clear`, in effect, sets the `Text` property to the Empty String, “” (no characters).

`outResults.Clear()`      ‘this line of code will clear the `outResults` rich text box  
                                 ‘this is equivalent to: `outResults.Text = “”`

Double clicking on the Quit button would add a place for its code directly under the code for the first button. The code needed for a *quit button* is simply the word `End`:

```
Private Sub btnQuit_Click(sender As System.Object, e As System.EventArgs) Handles btnQuit.Click
    'stop the run of the program
End
End Sub
```

It is good programming practice to put a quit button in each application. This gives the user an easy way to end the program at any time. It is also good programming practice to indent portions of the code in a program. This *indentation* allows a reader of the code to more easily understand the flow of logic within the program and identify which lines are part of the “body” of a task. In the programming environment you are using for Visual Basic 2013, most of the indenting is done for you automatically, but it is something to be conscious of as a programming practice that makes your code easier to read, understand, and maintain. The samples within this textbook demonstrate the use of indentation that you follow in writing your code.

### *Summary of good programming practices:*

1. Always rename objects after creating them using the naming conventions
2. Choose proper variable scope, (form-level vs local)
3. Always use comments in your code to describe what is being done
4. Use meaningful variable names
5. Declare your sub-routine variables at the top of your sub before other code
6. Use the auto-complete functionality (box that pops up occasionally offering suggestions)  
**Note:** press up and down arrows to scroll through auto-complete options and TAB to accept a completion. Using the enter key to accept completion will also move you to the next line.
7. Always have a Quit button on your form
8. Use Label objects to provide instructions on your forms
9. For easier editing, you can show line numbers in VB.Net: from the menu go to: Tools/Options/Text Editor/Basic/General/ and check the "Line numbers" box.
10. Save often

### *Saving Visual Basic Projects*

It is good practice to save your VB projects often. To do so, go to File > Save All or just click the Save All from the toolbar to save the changes since the most recent save. Here is an image of



the Save All icon:

Be sure to save your current program before trying to move on to the next one.

To run a saved program at a later time, open Visual Studio and click the Open File icon on the toolbar. Navigate to your desired folder and click on it. Open your program by clicking the “Microsoft Visual Studio Solution” version of your programName. It should have an icon with the Visual Studio logo and a “12” in the upper-right corner of the icon as shown in the following figure:

MyVBProjects ▸ SampleVBPrjobject		
Name	Date modified	Type
bin	1/6/2016 1:22 PM	File folder
My Project	1/6/2016 1:22 PM	File folder
obj	1/6/2016 1:22 PM	File folder
App	1/6/2016 1:22 PM	XML Configuration File
VB Form1.Designer	1/6/2016 1:23 PM	Visual Basic Source file
Form1.resx	1/6/2016 1:23 PM	RESX File
VB Form1	1/6/2016 1:23 PM	Visual Basic Source file
SampleVBPrjobject	1/6/2016 1:22 PM	Microsoft Visual Studio Solution
SampleVBPrjobject.v12	1/6/2016 1:23 PM	Visual Studio Solution User Options
SampleVBPrjobject	1/6/2016 1:23 PM	Visual Basic Project file

You can also open and run an existing program by choosing **File**, and then **Recent Projects and Solutions** or **Open Project** from the menu and then navigating to the desired project and selecting the project solution file you'd like to run.

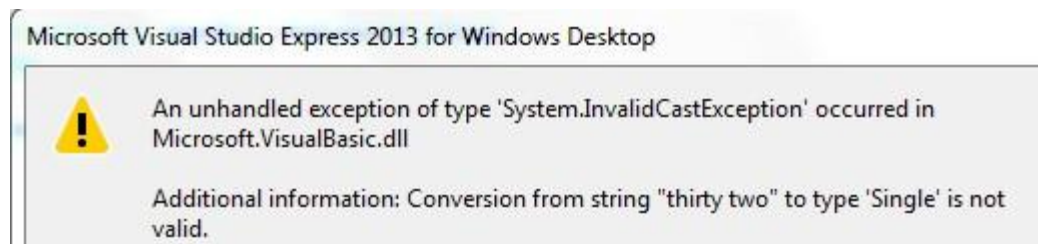
### 3.5 *Input & Output*

#### *Input Methods*

##### *Textbox*

Textboxes are used as a means of getting input from the user that will be used by the program. An assignment statement in the program assigns the user input to a variable. The assignment statement will automatically convert the text from the textbox into the appropriate data type. In the case of the temperature conversion program where the user is entering a Celsius temperature into a textbox, the assignment statement: `celsiusTemperature = txtCelsius.Text`, would automatically convert the user's input to a `Single` (decimal number) value before assigning it to the variable, `celsiusTemperature`.

If the user input cannot be converted to the correct data type, an error message will be displayed. (i.e. If the user typed in "thirty two", instead of 32, The program would be interrupted and the following message would be displayed:



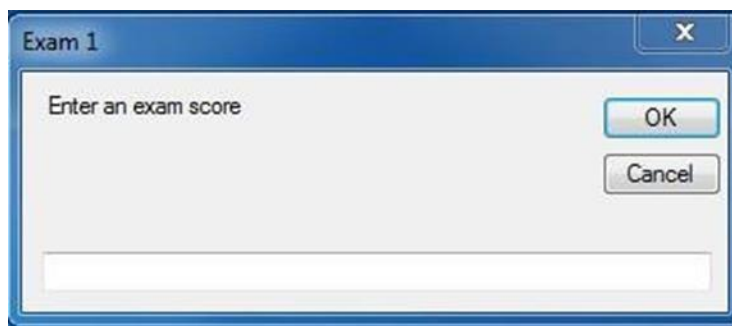
Textboxes are good for programs where the user need only input a few numbers or strings. If the user has many items to input, the form could get very cluttered with textboxes. If there is a lot of data as input, another method is preferable (see below).

### *InputBox Function*

Instead of using a textbox that is always on the form, the use of an InputBox function causes an input box to pop onto the screen as needed. Input boxes include a prompt and a title. The statement

```
score = InputBox("Enter an exam score", "Exam 1")
```

would produce the following input box:



Whatever the user types into the box will be assigned to the variable, score, when the user hits the OK button. Clicking OK also makes the input box disappear. If the user input cannot be converted to the desired data type, the program will be interrupted and an error message will be displayed, similar to the example for using Textboxes. If the user clicks on *Cancel* the program will also be interrupted.

### *Data Files*

Another solution to the problem of not wanting a lot of textboxes on your form is to use a separate data file for the data. Data files allow data to be previously entered and stored in a separate file. This way the data is always available and does not have to be entered by the user at run time. We will cover the use of data files in section 3.9.



## ***Output Methods***

### *RichTextBox*

The sample programs you have seen so far have used rich text boxes to display output. Section 3.4 in this text included an explanation of how to set the value of the Text property of a rich text box and how to append more text to it using the AppendText method. Here is a sample of each being used with a rich text box named *outResults*:

```
outResults.Text = celsiusTemperature & " degrees Celsius is equal to "  
outResults.AppendText(fahrenheitTemperature & " degrees Fahrenheit.")
```

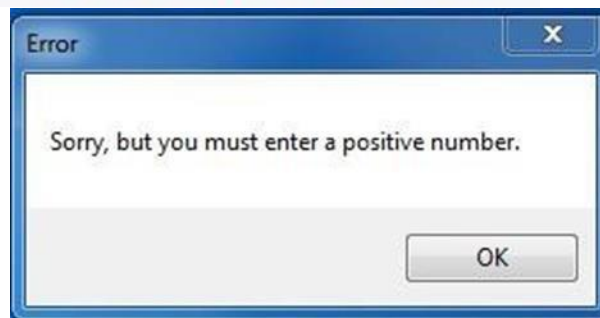
### *MessageBox Function*

Another way to display output, besides the rich text box, is to use a MessageBox function. Like the InputBox function, a MessageBox function also causes a box to pop onto the screen and are there only until you click OK to close them. This is particularly useful for error messages. A statement of the form:

```
MessageBox.Show(prompt, title)
```

pops up a box with the title on the top and the prompt inside. Both the prompt and the title should be Strings in quotation marks. The statement:

```
MessageBox.Show("Sorry, but you must enter a positive number.", "Error")
```



would produce the following output:

In the sample code in this chapter we use the ampersand, &, in many of our samples. The ampersand is a String concatenation operator that is used to join smaller Strings together into a larger String. Two or more strings with an & between them are joined into a single String. Note that joining Strings together does not introduce any spaces. If you want space in the resulting String, you can add it manually by concatenating a literal String of spaces, " ", to the desired String as shown on the next line where the assignment statement for myFullName demonstrates the manual addition of a space between the first and last name in the creation of myFullName.

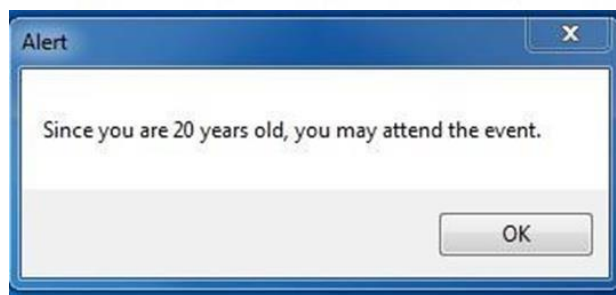
(ie. myFullName = myFirstName & " " & myLastName)



This operator is also often used to join a Strings with the values of one or more variables. This is especially useful in formulating output to be displayed that may consist of words and the values of variables. The & can also be used in assignment statements as was shown with myFullName,. There are other String operators available in VB, but the & is the only one used in this course.

Below is an example of the use of the ampersand to create an output message consisting of words joined with the value of a variable. If you had a variable, **age** that held someone's age, you could display the age as part of a message in a MessageBox.Show function as illustrated below. Note that the & symbol is used to join the pieces of the message together. The actual numeric value of the variable will be displayed when it is concatenated onto the message. In the sample below, showing the function call and the output, **age** has a value of 20.

```
MessageBox.Show("Since you are " & age & " years old, you may attend the event.", "Alert")
```



### *Formatting Output*

Sometimes when numeric information is displayed you would like to have it displayed in a certain format, possibly as currency if it is money or with a certain number of decimal places, or as a percentage. There are several format functions available to do that. While there are many possible formats to choose from, we will look at three of them here.

FormatCurrency formats a number as currency, adding a dollar sign, commas if needed, and as many positions after the decimal point as you specify.

```
Dim Balance as Single = 1254.3165
outResults.AppendText(FormatCurrency(Balance, 1))
$1,254.3      <= = printed result
```

```
outResults.AppendText(FormatCurrency(Balance, 0))
$1,254        <= = printed result
```

If you do not specify the number of decimal places, two is used by default.

```
outResults.AppendText(FormatCurrency(Balance))
$1,254.32     <= = printed result
```

FormatNumber takes a number in decimal form and rounds it off to the number of digits after the decimal point specified. Again, the default is two if you do not specify the number of places after the decimal point.

```
outResults.AppendText(FormatNumber(4.6389, 3))  
4.639 <= printed result
```

FormatPercent takes a number in decimal form and converts it to percent notation. As with FormatCurrency, you can specify the number of places past the decimal point, or use the default of 2. Extra zeroes are appended to match the desired number of decimal places in using any of these three formatting functions. All three formatting functions also use automatic rounding to determine the value with your specified number of decimal places. It is illustrated here using the FormatPercent function:

```
outResults.AppendText(FormatPercent(.4360295))  
43.60%
```

Note that you can use format functions as part of a larger string being displayed as is done below when displaying the temperature to three decimal places:

```
outResults.AppendText(FormatNumber(fahrenheitTemperature, 3) & " degrees Fahrenheit.")
```

### *Spacing*

One way to get the desired spacing in displayed output is to insert spaces into the enclosed quoted literal strings, as was done in the `outResults` line immediately above, but that approach is very tedious if you try to create a nicely aligned table with strings of varying length. When attempting to display information in table form, it is helpful to use a format string to assist with the exact alignment of the columns. A format string allows the programmer to allot the desired width for each column and also convey whether the alignment should be right-justified or left-justified. The format string first needs to be declared as a variable and initialized to the desired alignment specifications and then it is used in the line of code to display text in a rich text box. Below is an example of the use of a format string to display course names, buildings, and room numbers in a table format. The first `Dim` statement declares and initializes the format string. This format string indicates that there will be three data fields to be displayed. The first part of the format string, `{0,-25}`, indicates that the first field will be padded to be 25 characters in width, and the negative sign indicates the data will be displayed left-justified in that field. The second set of braces in the format string, `{1,-30}`, indicates that the second data item will be left-justified in a field of 30 characters and the `{2,15}` indicates that the third data item will be right-justified in a field 15 characters wide. You can enter as many sets of braces as needed, describing the width and justification for each data item.

```
Dim frmtString As String = "{0,-25}{1,-30}{2,15}"  
Dim course1 As String = "PRINTMAKING", building1 As String = "Art Building - SJU", room1 As String = "ART 020"  
outResults.AppendText(String.Format(frmtString, "Course Title", "Building", "Room") & vbNewLine)  
outResults.AppendText("*****" & vbNewLine)  
outResults.AppendText(String.Format(frmtString, "GRAPHICS", "Peter Engel Science Center", "PENGL 244") & vbNewLine)  
outResults.AppendText(String.Format(frmtString, course1, building1, room1) & vbNewLine)
```

Note that the format string can be used to display the values of variables, as was done in the last line of code, or the data item can be a literal string enclosed in quotes as the other two lines of code that use the format string demonstrate. The output from the prior code segment is:

Course Title	Building	Room
*****		
GRAPHICS	Peter Engel Science Center	PENGL 244
PRINTMAKING	Art Building - SJU	ART 020

Another thing to note about the use of a format string is that the expression used starts with `String.Format`. That syntax is different than a typical use of `.AppendText` where a format string is not being used. You may also want to note how the `vbNewLine` character was added to each line to place the next line added to the rich text box below the current one. Note: you can also add `vbNewLine` to the format string. This would prevent having to concatenate it to the end of each line. The code below demonstrates this and will produce the same output as above.

```
Dim frmtString As String = "{0,-25}{1,-30}{2,15}" & vbNewLine
Dim course1 As String = "PRINTMAKING", building1 As String = "Art Building - SJU", room1 As String = "ART 020"
outResults.AppendText(String.Format(frmtString, "Course Title", "Building", "Room"))
outResults.AppendText("*****" & vbNewLine)
outResults.AppendText(String.Format(frmtString, "GRAPHICS", "Peter Engel Science Center", "PENGL 244"))
outResults.AppendText(String.Format(frmtString, course1, building1, room1))
```

### 3.6 Numeric and String Functions

#### Numeric Functions

Besides the usual arithmetic operations, many standard mathematical functions are built into Visual Basic. Some of the functions you might wish to use include:

`Math.Sqrt(X)` returns the square root of X

Sample use of the function:

```
outResults.AppendText(Math.Sqrt(4))
```

results produced:

2

`Int(X)` truncates X, removing any part past the decimal point

Sample use of the function:

```
outResults.AppendText(Int(4.83))
```

results produced:

4

`Math.Round(X)` rounds X off to the nearest integer

Sample use of the function:

```
outResults.AppendText(Math.Round(4.8))
```

results produced:

5

Math.Abs(X) returns the absolute value of X

Sample use of the function:

```
outResults.AppendText(Math.Abs(-4))
```

results produced:

4

As an example of a program using a numeric function, suppose you wished to find the two roots of a quadratic equation. The formula for finding these roots is

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
Private Sub btnQuadratic_Click(sender As System.Object, e As System.EventArgs) Handles btnQuadratic.Click
    'this program will calculate the roots of a quadratic equation
    'declare variables
    Dim a As Integer = 0, b As Integer = 0, c As Integer = 0
    Dim r1 As Double = 0.0, r2 As Double = 0.0

    'get values of coefficients from user input
    a = txtA.Text
    b = txtB.Text
    c = txtC.Text

    'calculate roots to the equation
    r1 = (-b + Math.Sqrt(b ^ 2 - 4 * a * c)) / (2 * a)
    r2 = (-b - Math.Sqrt(b ^ 2 - 4 * a * c)) / (2 * a)

    'display the roots
    outResults.AppendText("The two roots of the equation are: " & r1 & " and " & r2)
End Sub
```

### *String Functions*

There are also several built in functions for manipulating strings. String functions can appear in output statements or their results can be assigned to a string variable.

Left, Right, and Substring return a substring from the left side, right side, or middle of a string.

sample use of the function: `outResults.Text = (Strings.Left("Saint John's", 7))`  
'note the use of the Strings prefix

results produced: Saint J

sample use of the function: `outResults.Text = (Strings.Right("Saint Ben's", 5))`  
'note the use of the Strings prefix

results produced: Ben's

sample use of the function: `outResults.Text = ("St. Joseph".Substring(4, 4))`  
'start at index 4 and take 4 characters  
'note that the index starts at 0  
'note the absence of Strings when using Substring

results produced: Jose

These functions can also be used with String variables, instead of the literal strings used in the samples given. Here is a segment of code that demonstrates using the Left, Right, and Substring functions with variables. Note the concatenation of *vbNewLine* near the end of the two lines following: sample use of the functions:

```
Dim yourTown As String = "Smallville"
outResults.AppendText(Strings.Left(yourTown, 5) & vbNewLine &
    Strings.Right(yourTown, 8) & vbNewLine &
    yourTown.Substring(1, 4))
```

results produced: Small  
allville  
mall

Some String functions return a number rather than a string. The IndexOf function searches a string for a character or substring and returns the index of the first position it is found. If it is not found, -1 is returned. This function takes two parameters, a string to search for and the index to start the search at.

sample use of the function:

```
Dim positionOfComma As Integer = 0, startPosition As Integer = 0
'look for a comma and return the position in the string where it was found
'** Remember that the index of the first character is zero
positionOfComma = "Avon, MN".IndexOf(",", startPosition)
outResults.AppendText("A comma was found at index " & positionOfComma & vbNewLine)
```

results produced: A comma was found at index 4

Some String functions are used as if they were properties of a String object, much like the Text property is used for text boxes. You can access the value of these attributes by referring to the property using the name of the String variable and the property name. You can assign the property value to variables, use it in an expression, or display it in an output box.

The Length function returns the length of a string, including spaces.

sample use of the function:

```
Dim yourTown As String = "Avon Springs"
outResults.AppendText("The length of the name of your town is: " & yourTown.Length)
```

results produced: The length of the name of your town is: 12



The Trim function returns a copy of the original string with all of the spaces removed from the beginning and end of the String. The Trim function does not change the original String.

sample use of the function:

```
Dim yourTown As String = "  Avon Springs  "
outResults.AppendText("Your town is:" & yourTown.Trim & ", MN!")
```

results produced: Your town is:Avon Springs, MN!

The ToUpper function returns a copy of the String with all of the characters in UPPER CASE. It does not change the original String. (There is a similar function, ToLower.)

sample use of the function:

```
Dim yourTown As String = "Avon Springs"
outResults.AppendText("Your town is: " & yourTown.ToUpper)
```

results produced: Your town is: AVON SPRINGS

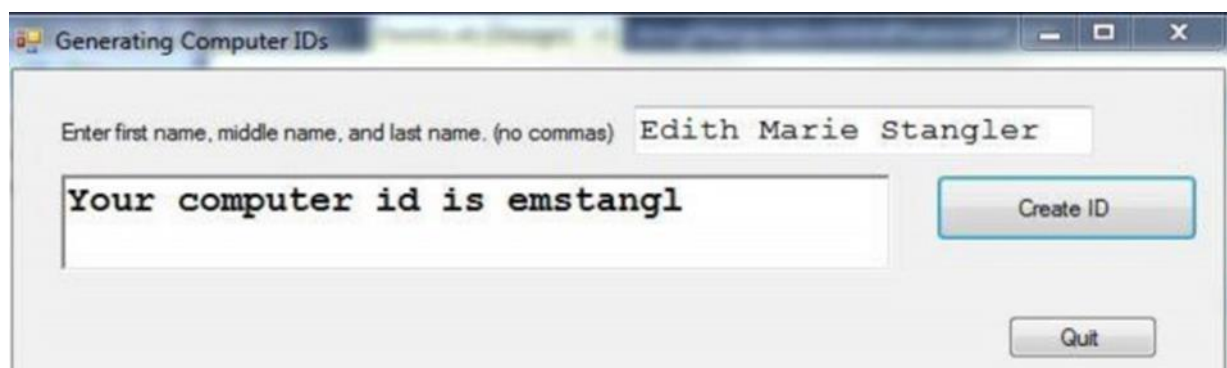
Some String functions return a Boolean value, either True or False. One that you may wish to use is the Contains function. This function takes a target string as a parameter returns True if the original string contains the target string and False if it does not contain it.

sample use of the function:

```
Dim yourTown As String = "Avon Springs"
outResults.AppendText("It is " & yourTown.Contains("ring") &
    " that " & yourTown & " contains the word 'ring'.")
```

results produced: It is True that Avon Springs contains the word 'ring'.

The following example program uses these string functions to take a person's first name, middle name, and last name as input and return a computer id name composed of that person's first and middle initials and the first six characters from the last name, all in lower case.



```

Private Sub btnCreateID_Click(sender As System.Object, e As System.EventArgs) Handles btnCreateID.Click
    'declare necessary variables for this subroutine
    Dim wholeName As String = "", id As String = ""
    Dim firstSpaceLocation As Integer = 0, secondSpaceLocation As Integer = 0
    Dim firstName As String = "", middleInitial As String = "", lastName As String = ""
    Dim wholeNameLength As Integer = 0

    outResults.Clear()
    wholeName = txtName.Text

    'use the Length property to get the length of a string
    wholeNameLength = wholeName.Length

    'indexOf starts with index of 0
    firstSpaceLocation = wholeName.IndexOf(" ", 0)

    'extract the first name from the whole name using the Left function
    firstName = Strings.Left(wholeName, firstSpaceLocation)

    'get middle initial, just after the first space
    middleInitial = wholeName.Substring(firstSpaceLocation + 1, 1)

    'calculate index of second space, start searching after the first space
    secondSpaceLocation = wholeName.IndexOf(" ", firstSpaceLocation + 1)

    'extract the last name
    lastName = Strings.Right(wholeName, wholeNameLength - secondSpaceLocation - 1)

    'create the ID from the three pieces and change it to lower case
    id = Strings.Left(firstName, 1) & middleInitial & Strings.Left(lastName, 6)
    id = id.ToLower

    'output the result
    outResults.AppendText("Your computer id is " & id & vbNewLine)

End Sub

```

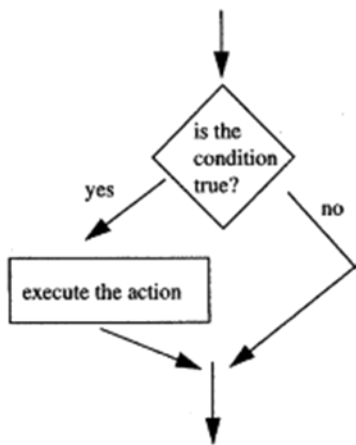
Notice that all variables are declared. After the declarations we clear the output box of any previous output. We then search for the first space, between the first name and middle initial. The location of this space is assigned to the variable `firstSpaceLocation`. The variable `firstName` is assigned everything to the left of `firstSpaceLocation`, in other words, the first name. The variable `middleInitial` is assigned the 1 character to the right of `firstSpaceLocation`, the middle initial. In order to know how long the last name is, we have to find the location of the second space. To find the second space, note that we had the `IndexOf` function start the search at the index just to the right of the first space that was found (instead of at index 0 as we did when looking for the index of the first space). Then we can use the `Right` function in conjunction with the location of the second space to extract the last name from the whole name. The final id is a concatenation of the first letter of the first name, `middleInitial`, and the first six letters of the last name. If there aren't six letters in the last name, that is not a problem as the `Left` function will then just return the entire string.

### 3.7 Conditional Statements

Sometimes a programmer wants to execute an action only under certain conditions. In VB, there is a programming construct called an **If statement** that gives the programmer control over when some sections of code will be executed.

#### If Statements

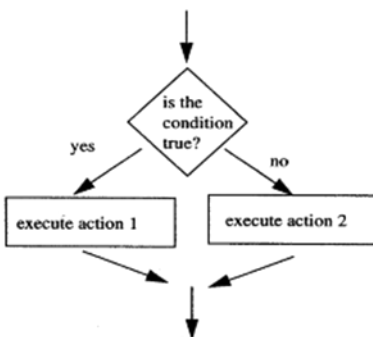
If statements are used in two different circumstances. Sometimes you will want an action to be performed only when a certain condition is met, with nothing happening otherwise.



If you need to execute code only when a certain condition is met use this Simple-If form:

```
If <condition> Then
    <code to be executed if condition is true>
    <optionally more and more code>
End If
<code to be executed regardless of condition>
```

Sometimes one action is desired when the condition is true and a different action when the condition is false as diagrammed below.





If you need to execute code when a certain condition is met and different code if the condition is not met use the If-Then-Else form:

```
If <condition> Then
    <code to be executed if condition is true>
    <optionally more and more code>
Else
    <code to be executed if condition is false>
    <optionally more and more code>
End If
<code to be executed regardless of condition>
```

The conditions must evaluate to an expression that is true or false, also called a Boolean expression. You can use the following logical operators to create these conditions:

operator		sample use
=	is Equal to	e.g.: If GPA = 4.0 Then
<>	is Not Equal to	e.g.: If major <> "CSCI" Then
>	is Greater Than	e.g.: If score > 100 Then
<	is Less Than	e.g.: If selfEsteem < 0 Then
>=	is Greater Than or Equal To	e.g.: If bankBalance >= 1000 then
<=	is Less Than or Equal To	e.g.: If temperature <= 0 Then

Using logical operators like <= makes intuitive sense when you are comparing two numeric values. It is less obvious what that might mean when comparing two Strings. In VB, string comparison is done by comparing the characters within each string, from left to right. You can think of the letters of the alphabet, a,b,c, ... as being represented by a list of numbers where 'a' is a number that is smaller than 'b' and 'b' is smaller than 'c'. Using this type of scheme, VB can then compare two Strings for equality if the 'numbers' used for each character in the two strings are exactly the same. In a similar fashion we could say that "talk" < "walking" since the first letter of "talk" is less than the first letter of "walking" and since the comparison goes left to right, as soon as VB compares the 't' with the 'w', it discovers that the first string is "less than" the second one. All characters have an associated value, so strings that contain characters other than letters are compared in the same way. If the first character of both strings are identical, the next character is checked. There is more information about the details of how characters are represented in the chapter on data representation.

Complex conditions can be created using parentheses and the relational operators, NOT, AND, and OR. The precedence rules for evaluating a logical expression are similar to evaluating arithmetic expressions where the parentheses are looked at first and can be used to change the order of evaluation of the expression. NOT has the highest precedence, followed by AND and then OR. The Boolean negation operator, NOT, is used to reverse the meaning of a Boolean expression, (ie. *Not ( bankBalance < 100)* would mean the balance is at least 100.)

The following example explores how the use of parentheses can be used to change the meaning of a Boolean expression:

```
If major = "CSCI" AND year = "Senior" OR year = "Junior" Then
```

This is the same as:

```
If (major = "CSCI" AND year = "Senior") OR year = "Junior" Then
```

We probably wanted this though:

```
If major = "CSCI" AND (year = "Senior" OR year = "Junior") Then
```

Note that using an equal sign in an If statement is NOT the same as the use of an equal sign in an assignment statement. "If GPA = 4.0" does not assign 4.0 to the GPA variable, it is just asking the question "Is GPA equal to 4.0?"

Consider the following examples that use the If statements. A customer wishes to withdraw a certain amount of money from her bank account. The bank wishes to allow the withdrawal only if there is enough in the account to cover the amount. The code for the withdrawal button might look as follows:

```
Private Sub btnWithdrawal_Click()  
    Dim amount As Single = 0.0  
    Dim balance As Single = 0.0  
    balance = txtBalance.Text  
    amount = txtAmount.Text  
    If balance > amount Then  
        balance = balance - amount  
    End If  
    outResults.AppendText("Your new balance is " & FormatCurrency(balance))  
End Sub
```

Notice that if the amount the user wishes to withdraw were greater than their balance the old balance would be output as a new balance. This is not very helpful. We could use the following code to print a new balance only if the condition was true:

```
    If balance > amount Then  
        balance = balance - amount  
        outResults.AppendText("Your new balance is " & FormatCurrency(balance, 2))  
    End If
```

The code above would not display anything at all if the user tries to withdraw more than is in the account. It would be much better to warn them of this in some way. In other words, we would like a specific action to be executed when the condition is true and a different action to be executed when the condition is false. The structure to handle this situation is the If-Then-Else variation of the If statement.

This structure uses an If statement with an Else clause added to it to execute one set of actions for a true condition and a different set of actions for a false condition:

```
If balance > amount Then
    balance = balance - amount
    outResults.AppendText("Your new balance is " & FormatCurrency(balance, 2))
Else
    outResults.AppendText("You cannot withdraw " & FormatCurrency(amount, 2) & vbCrLf)
    outResults.AppendText("You only have " & FormatCurrency(balance, 2) & " in your account")
End If
```

Notice the indentation conventions used. Only the If line, Else line, and the End If extend out to the current margin and the actions are indented. This makes the structure of the statement easy to see at a glance.

Let's look at one more example that includes more than two conditions. Suppose a user wishes to purchase a number of notebooks from the bookstore. The notebooks are \$1.75, but if you buy more than twenty, you receive a thirty percent discount, if you buy from fifteen to twenty you get a twenty percent discount, and if you buy more than 10, (but less than 15), you get a ten percent discount. Sales tax is 6% and is applied after the discounts have been taken.

There are 4 conditions on the number of notebooks:

1.  $\leq 10$
2.  $> 10$  and  $< 15$
3.  $\geq 15$  and  $\leq 20$
4. Over 20

To handle problems with more than two conditions there is another variation of the If statement called the If-Then-ElseIf statement that is useful and requires only one End If statement instead of the several that would be needed if multiple simple If statements were used. Using an If-Then-ElseIf statement is preferable to using several simple If statements. If you would use several simple If statements, every one of the conditions would have to be evaluated when the code is executed, while in an If-Then-Else statement the program exits the entire If construct as soon as it completes the action(s) for the first condition that is true. This is both more efficient and requires fewer lines of code. Following is a solution to the problem stated above that uses the IfThen- ElseIf statement to handle multiple conditions:

```

Private Sub btnNotebookBill_Click(sender As System.Object, e As System.EventArgs) Handles
    'this will calculate the cost of notebooks purchased
    Dim notebooks As Integer = 0, totalBill As Single = 0.0, discountRate As Single = 0.0
    Dim salesTaxRate As Single = 0.06, subtotal As Single = 0.0, myMessage As String = ""
    Dim tax As Single = 0.0, price As Single = 0.0
    price = 1.75
    notebooks = txtQuantity.Text
    myMessage = "Your total is "

    If notebooks <= 10 Then           'condition 1
        discountRate = 0
    ElseIf notebooks < 15 Then       'condition 2
        discountRate = 0.1
    ElseIf notebooks <= 20 Then      'condition 3
        discountRate = 0.2
    Else
        discountRate = 0.3           'for all other conditions
    End If
    subtotal = (notebooks * price * (1 - discountRate))
    tax = subtotal * (salesTaxRate)
    totalBill = subtotal + tax
    outResults.AppendText(myMessage & FormatCurrency(totalBill, 2))
End Sub

```

### *Nested If Statements*

ifs can be nested (one inside another)—you will occasionally need to nest a complicated if statement inside another. Example:

```

If major = "CSCI" Then
    outResults.AppendText("Hello CS Major!")
    If zipcode.Contains("7") And studentName.Length > 10 Then
        outResults.AppendText("I see you have a 7 in your zip and a long name")
    End If
Else
    outResults.AppendText("I only say cool stuff about CS Majors")
End If

```

The above example will first determine if major = “CSCI” is true. If so, it will check the inner if statement. If major = “CSCI” is false, the inner if statement will never be evaluated. The program would jump immediately to the output statement in the Else clause.

### 3.8 *Loops*

Often in the solution to a problem, there is a need to repeat certain actions. VB has several loop structures to handle repetition and we will examine two of them, the For...Next loop and the Do While loop. Every loop needs three parts to it. There must be a condition to determine when the loop will end. There also must be some initial state for the variable that determines that condition. Finally, there must be some action(s) to be repeated.

#### **For...Next Loops**

When we know exactly how many times we wish a loop to repeat, we can use a For...Next loop, (also referred to as a For loop.) This type of loop includes both the initialization of the variable and the condition on which to stop looping in a single line. The general form of the For...Next loop is:

**For** *controlVariable* = <initial value of control variable> **to** <final value of control variable>  
  
    < loop body, the statements/actions to be repeated >

#### **Next** *controlVariable*

To demonstrate and use as an example to explain the workings of the For...Next loop, here is a program that uses a For...Next loop to repeat an action five times. The action is a trivial one, just print the word “hi” five times. (something like a high-five, only “hi” five ☺ )

```
Private Sub btnHiFive_Click(sender As System.Object, e As System.EventArgs) Handles btnHiFive.Click
    'declare variable
    Dim hiCounter As Integer = 0

    'display 'hi' five times
    For hiCounter = 0 To 4

        outResults.AppendText("hi" & vbNewLine)

    Next hiCounter

    outResults.AppendText(" 'hi' was printed " & hiCounter & " times.")
End Sub
```

Intuitively you can think of using the variable hiCounter to keep track of how many times you have printed the word “hi”. Intuitively, after you print the first “hi”, hiCounter should be 1, after the second “hi”, hiCounter should be 2, etc. Usually when you start counting items you begin with 1, but computer scientists have quite a habit of starting with 0 instead, (kind of weird, huh?). There are reasons you will come to understand after further study. In the first line of the For Loop in the program above, the value of hiCounter is initially set to 0; then when the line of code:

Next hiCounter

is executed, the value of `hiCounter` is automatically incremented by one and control goes back to the first line of the loop (the `For` statement) where the value of `hiCounter` is checked to see if it is still in the range between 0 and 4. As long as the value of `hiCounter`, the loop control variable, is in the range between the initial value and the final value in the `For` statement, the loop will repeat. The loop will terminate only when the value of `hiCounter` takes on the value five (after the loop has executed five times). Note the values of `hiCounter` while the loop is executing run from 0 to 4, and that the series 0,1,2,3,4 has the same number of elements as the series 1,2,3,4,5 that we might use if we were counting the number of times “hi” was printed. Since `For` Loops are usually used when you know how many repetitions you want, say  $n$  times, you will often use `For` Loops that run from 0 to  $n-1$ , effectively repeating  $n$  times. Note that a side benefit of starting `hiCounter` at zero is that when the loop terminates, the value of `hiCounter` is 5, the number of times “hi” was printed and the number of times that the actions within the loop were repeated. We can then use the variable `hiCounter` later in our program as was done in the given code, if needed.

Here is another `For` Loop sample that would print the numbers from 1 to 100.

```
For number = 0 To 99
    outResults.AppendText(number + 1 & vbCrLf)
Next number
```

The variable, *number*, starts at 0. When the `Next` statement is encountered 1 is automatically added to *number* and the ending condition is checked. When the program reaches “`Next number`”, it loops back to the `For` statement and checks the condition to see if *number* is still within the defined range, 0 to 99. The program continues while *number*  $\leq$  99.

### Steps

A simple `For...Next` loop automatically increments the control variable by 1. But we can change this by adding the appropriate step. Steps can be any positive or negative number, integer or real. For example, if we wished to print the even numbers from 100 down to 2 we would use a step of -2:

```
For downByTwo = 100 To 2 Step -2
    outResults.AppendText(downByTwo)
Next downByTwo
```

You can use variables to replace any of the numbers:

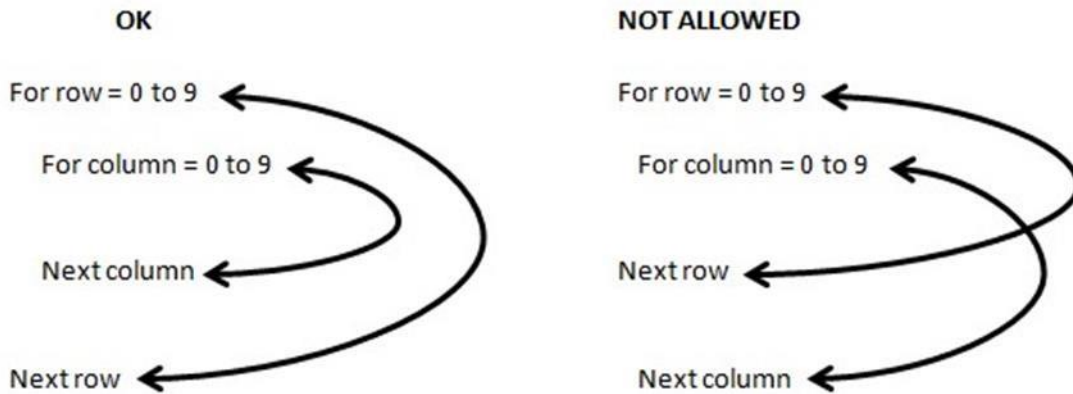
```
myStart = InputBox("Where should I start?", "Start?")
myFinish = 100 + myStart
myStep = InputBox("By How Much?", "Step?")

For mySeries = myStart To myFinish Step myStep
    outResults.AppendText(mySeries)
Next mySeries
```



## Nested Loops

Loops may not overlap. However, one loop may be completely embedded or nested inside another.



When one loop is nested inside another, as on the left in the diagram above, the inner loop will be executed completely for each execution of the outer loop. In other words, in the above example, the column loop will be executed ten times for each execution of the row loop, for a total of 100 times.

The following program uses nested loops and generates a multiplication table. Each row in the table has 10 entries and there are 10 rows, for a total of 100 entries. The row loop moves the output from one row to the next while the column loop calculates and displays the ten entries in each row. Following is the program.

```
Private Sub btnMultiplicationTable_Click(sender As System.Object, e As System.EventArgs)
    Dim row As Integer = 0, column As Integer = 0, frmStr As String = "{0,6}"
    outResults.AppendText("Multiplication Table" & vbNewLine)
    outResults.AppendText(vbNewLine) 'prints a blank line

    For row = 0 To 9

        For column = 0 To 9
            outResults.AppendText(String.Format(frmStr, (row + 1) * (column + 1)))
        Next column

        outResults.AppendText(vbNewLine) 'moves the cursor to the next line
    Next row
End Sub
```

The output from this program would be:

Multiplication Table

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

## Do While Loops

Do While Loops use a conditional statement to control the repetition. As long as the condition is “true” the loop activities will be repeated. The condition is evaluated each time through the loop and, as long as (WHILE) the condition is true, the repetition continues. If the condition is initially false, the loop is never executed, the loop is skipped over, and the lines below the loop will be the next code to be executed.

The general format of a Do While loop is:

**Do While** < condition >

< loop body, the actions/statement(s) to be repeated while the condition is true >

< to avoid an infinite loop, these statements must include a statement > <  
that will eventually make the condition false >

## Loop

The condition in a Do While Loop is a Boolean condition (evaluates to True or False) involving the loop control variable. Below is a sample program that demonstrates the use of a Do While Loop to print out the word “hi” five times.

```
Private Sub btnHi5WhileLoop_Click(sender As System.Object, e As System.EventArgs)
    'declare variable
    Dim hiCounter As Integer = 0

    'display 'hi' five times
    Do While hiCounter < 5
        outResults.AppendText("hi" & vbNewLine)

        'increment the loop control variable
        hiCounter = hiCounter + 1
    Loop

    outResults.AppendText(" 'hi' was printed " & hiCounter & " times.")
End Sub
```

When using a Do While Loop, we need to first initialize the variable that controls the repetition of the loop. This initialization occurs outside the loop. In the program above the initialization is



part of the Dim statement for the loop control variable for this program, hiCounter. The condition controlling the loop is part of the Do While statement. The actions to be repeated are in the body of the loop (the lines between Do and Loop) and the loop ends with the word Loop, which loops the program back to check the condition and if it is still true, continue another repetition of the body of the loop. Note the line in the prior program that increments the loop control variable, hiCounter. This line is needed to eventually cause the loop to terminate. Without this line, the value of hiCounter would never change from its initial value of zero and the condition, hiCounter < 5, would remain true forever and the program would be in an infinite loop. Press the red stop button in Visual Studio to exit an infinite loop. Here is another sample of a Do While Loop:

```
Private Sub btnDowhileSample2_Click(sender As Object, e As EventArgs) Han
    'This program will display the multiples of ten from 100 down to 50
    'and also calculate and display their sum.

    'declare variables and set initial values of each
    Dim sum As Integer = 0, multipleOfTen As Integer = 100

    Do While multipleOfTen >= 50
        'display each multiple of ten
        outResults.AppendText(multipleOfTen & vbNewLine)

        'add each multiple to the sum
        sum = sum + multipleOfTen

        'set the value of the next smaller multiple of ten
        multipleOfTen = multipleOfTen - 10
    Loop

    'display the sum of the multiples of ten from 100 down to 50
    outResults.AppendText("The sum of these multiples is " & sum)
End Sub
```

The output from the previous code is listed here:

```
100
90
80
70
60
50
The sum of these multiples is 450
```

### *Sentinel Loops*

While loops are especially good when a user is in control of when to end the loop. In other words, there are times when the programmer may not know how many times the loop should be repeated. Suppose we wanted a program that would add lists of numbers of any size, with the lists entered by the user. The user could signal the end of the list by entering a predetermined flag, a number that one would not expect to be on the list. If we were averaging scores on an

exam, for example, one might use -1 as a flag, since one cannot get a negative score. Common flags are -99 and -999.

The algorithm for such a program would be:

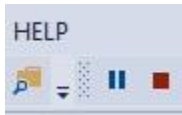
1. Enter a grade
2. If grade  $\neq$  -1 then
  - add grade to sum
  - add 1 to count
  - enter another grade
3. Repeat step 2 as long as grade  $\neq$  -1
4. Divide sum by count to obtain the average
5. Print average

```
Private Sub btnAverage_Click(sender As System.Object, e As System.EventArgs) Handles  
    'program to average exam scores in a list of any size  
    Dim grade As Single = 0.0, sum As Single = 0.0, count As Integer = 0  
    Dim avg As Single = 0.0  
  
    grade = InputBox("Enter the first exam score.", "Exam Scores")  
  
    Do While grade <> -1  
        'add grade to sum  
        sum = sum + grade  
        'increment count to keep track of how many scores were entered  
        count = count + 1  
        grade = InputBox("Enter another score. Type -1 to end.", "Exam Scores")  
    Loop  
  
    'calculate the average score and display the result  
    avg = sum / count  
    outResults.AppendText("The average on this exam is " & avg)  
End Sub
```

### *Tips for Using Loops*

- Take your time and figure out the condition and control to exit the loop
- IMPORTANT: If your loop is not producing the desired results, use rich text box print statements or `MessageBox.Show()` inside your loop to see what's going on each time through. Printing the value of your control variable or other variables inside the loop can often tell you what's wrong, or maybe you'll find out that the loop is not even executing!

- Infinite loops will happen – just need to stop your program and fix it. TO STOP AN INFINITE LOOP PRESS THE STOP BUTTON (red square to the right of HELP on the tool bar.



- When you use Do While Loops, double-check your logic to make sure the condition you choose is correct.

### 3.9 Using Data Files

We can also use Do While Loops with data files. To create a data file to use as input for a program, open the Notepad text editor on your computer (from the lower task bar, click Start, All Programs, Accessories, and then Notepad). Enter your data on the notepad, separating individual pieces of data with commas or by placing them on separate lines. If you have a data item that contains a comma, put that entire string in quotation marks. Here is some sample data that has been put into a notepad file called “QuizScores.txt” that will be used below in a program to illustrate how to use a file as input in a program.

John Smith, 10, 9, 10  
 Mary Engelman, 8, 10, 9  
 Andrew Bailey, 8, 7, 10  
 “Jacobson, Suzanne”, 9, 7, 9

When all of your information has been entered, save the data file with its own name. Be sure not to include any blank lines at the end of your file. You can now use this data in your program.

Before you can read data from a file or write data to a file in your VB program, you have to first open the file (ie. `FileOpen(1, "QuizScores.txt", OpenMode.Input)` which establishes a connection to the file on a channel number, channel number ‘1’ in this sample, that you then use when you read from or write to it. The “OpenMode.Input” specifies that the file is being opened to be read rather than written to. When reading data from a data file, the items are read one at a time, using an Input statement, (ie. `Input(1, quiz3)`). The Input statement gets the next item from the file and assigns its value to the variable (in this sample, that variable is `quiz3`). A special end of file function, `EOF(1)`, that has a value of either true or false is used in the condition of the while loop that controls the repeated reading of the data. The EOF function checks to see if you are at the end of the file on the specified channel and returns either a value of true or false each time you use the function. By incorporating the end of file function into the condition of the while loop as shown here: **Do While Not EOF(1)**, the code in the loop will repeatedly go and get the next data item until the end of the file is reached. You are essentially telling the computer to keep reading data from the file on channel number one as long as you have not reached the end of the file. At the end

of the program you must include a `FileClose(1)` statement to close the file on channel 1 (or whatever channel is used).

When using a `FileOpen` statement to access a data file, the program needs to know where to find the file. When the data file is stored in the `bin/Debug` folder of your VB project, you can indicate the path to the file by simply using the file name. If your data file is not stored in that location, you would need to type in the drive letter and name of all of the folders in the complete path to your file. In this course, you should place your data files in the `bin/Debug` folder. This will allow your programs to work when you hand in your VB project folders to be graded without having to change your code to reflect a new path for your file.

Shown next is a sample program that uses the data file named “QuizScores.txt” and introduces the statements needed to read data from a file. The purpose of the program is to calculate and print the average quiz score for every student in the file. The data file includes the student name followed by three scores for each one. The data file looks like this:

```
Maria Engelman, 10, 9, 8
Andy Ramirez, 8, 8, 10
Luke Skywalker, 9, 10, 10
```

```
Private Sub btnQuizAverage_Click(sender As Object, e As EventArgs) Handles btnQuizAverage.Click
    'this program will calculate and print the average quiz score for each student
    'the information needed is input from a data file
    Dim avg As Single = 0, student As String = ""
    Dim frmtString As String = "{0,-25}{1,10}" & vbCrLf
    Dim q1 As Integer = 0, q2 As Integer = 0, q3 As Integer = 0

    'file must be in bin/Debug folder for this project
    FileOpen(1, "QuizScores.txt", OpenMode.Input)

    'display header for table of student quiz score averages
    outResults.AppendText(String.Format(frmtString, "Student Name", "Quiz Average"))

    Do While Not EOF(1)
        'get input from the file for one student
        Input(1, student)
        Input(1, q1)
        Input(1, q2)
        Input(1, q3)
        'calculate the average score
        avg = (q1 + q2 + q3) / 3
        'display results for this student as the next entry in the table
        outResults.AppendText(String.Format(frmtString, student, FormatNumber(avg, 1)))
    Loop
    FileClose(1) 'close the data file on channel 1
End Sub
```

Following is the output from the program above:

Student Name	Quiz Average
Maria Engelman	9.0
Andy Ramirez	8.7
Luke Skywalker	9.7

The following program demonstrates both reading from and writing to a file. The program reads city names and corresponding populations from a data file containing all of the cities of Minnesota and writes the names of the cities and the population of the cities having populations under 200 to two separate files. One of the output files is written using the WriteLine statement and the other was written using the PrintLine statement. Comments in the code explain the difference and a portion of the resulting files are listed after the code.

```
Private Sub btnReadWriteFiles_Click(sender As System.Object, e As System.EventArgs) Han
    Dim cityName As String = "", cityPopulation As Long = 0
    FileOpen(1, "MinnesotaCitiesAndPopulations.csv", OpenMode.Input)
    FileOpen(2, "MinnesotaCitiesWithPopulationUnder200.txt", OpenMode.Output)
    FileOpen(3, "SmallMinnesotaCitiesReport.txt", OpenMode.Output)

    'print a header for this report file, this will be the first line of the file
    PrintLine(3, "Minnesota Cities with Populations Under 200 ... as of 2010")

    Do While Not EOF(1)
        'get the data from the file
        Input(1, cityName)
        Input(1, cityPopulation)

        If cityPopulation < 200 Then
            'Write or WriteLine are used to create files that may be used as data
            'in another program.
            'Notice the commas that are inserted in the file between items
            WriteLine(2, cityName, cityPopulation)

            'Print or Printline are used to print information to a file, like a report
            'notice there are no commas in the output
            PrintLine(3, cityName, cityPopulation)
        End If
    Loop
    FileClose(1)
    FileClose(2)
    FileClose(3)
End Sub
```

Here are the first few lines from the output file created using the WriteLine statement:

```
"Alberta",103
"Aldrich",48
"Alpha",116
"Angle Inlet",60
```



Here are the first few lines from the output file created using the PrintLine statement:

```
Minnesota Cities with Populations Under 200 ... as of 2010
Alberta      103
Aldrich      48
Alpha        116
Angle Inlet  60
```

### *Tips for Using Data Files*

- Commas in files have special behavior—they are treated as a separator between data items when reading in the data from a file. In the data file used in a previous example, a comma separates the student’s name from the first score and also is used between each adjacent pair of scores. When the string data contains a comma, it is enclosed in double quotes; numbers are without quotes. Watch out for missing commas!
- Don’t forget to Close your files when you are done!
- Do NOT have any blank lines in the file, even at the end—you will get errors! Create data files with Notepad (do NOT use Microsoft Word, it saves files in a format that cannot be read by Visual Basic). Actually it’s even a bad idea to copy/paste from Word as quote characters sometimes don’t copy over correctly.
- Note, commas are only needed if each line in your file contains multiple pieces of data (multiple fields). Each line can be thought of as either a single “item” where no commas would be needed or a “record”, where the related items would be separated by commas. A comma can be inside the data as long as it is in quotes: “Trutwin, Josh” is ok. 4,000,000 (the number 4 million) is NOT ok as it will read only the number 4. Use 4000000 instead.
- Booleans cannot be placed in a file—though rarely is there need to.

### **3.10 Arrays**

So far the variables that we have used point to a single location in memory and hold a single value. There are times, however, when we want to store lists of related data in the computer. Suppose, for example, we wish to store the times for 75 runners competing in a marathon. We would not want to use 75 separate variables for this task.

An array is a variable that holds multiple pieces of data, such as a list or a table and acts as a convenient way to refer to a collection of data items. You refer to the individual objects in the array by their position in the list. The particular position in the list is in effect a unique variable that holds one of the items in the collection. The position of an item that determines its location in the list is often referred to as the item’s subscript, or sometimes its index. The first position in the list is position 0 or subscript 0, the second position is 1, the third position 2 . . . . Computer scientists have the habit of starting their counting at zero instead of one. Reasons for starting with zero will become more clear when you study data representation in Chapter 6. When you

declare an array you give it a name, the value of the highest subscript and the data type of the collection of items.

The arrays we will be using in this course have a predetermined size that does not change during the program or from run to run and are called static arrays. To declare a static array, you must give the name, highest subscript, and data type using the following format for the Dim statement:

**Dim** *arrayName(highestSubscript) As DataType*

For example, two arrays to hold the names of the runners entered in a race and their times might be declared as follows.

```
Dim runner(75) As String
Dim time(75) As Single
```

This sets up two lists with 76 slots in each, (remember the first position is subscript 0). The runner array will hold up to 76 Strings and the time array will hold up to 76 decimal numbers. The first few slots of the runner array are shown below.

runner(0)	runner(1)	runner(2)	runner(3)	runner(4)	runner(5)	.....
-----------	-----------	-----------	-----------	-----------	-----------	-------

If you wanted to print the name of the 5th runner, you would use the code:

```
outResults.AppendText(runner(4))
```

To fill an array, change its contents, or print its contents, it is best to use a loop. The following code segments give the basic method for each of these procedures based on the arrays *runner* and *time*, declared above, that are used to hold each runner's name and time.

*Filling two arrays from a data file:*

```
'Declare arrays and a variable for counting the number of elements in the arrays
Dim runner(75) As String
Dim time(75) As Single
Dim numElements As Integer = 0
'note that numElements was initialized to zero in the Dim statement above
'it will be used to count the array items and also as the position within the array

'Prepare the file to be read, (open it)
'NOTE: text file must be in this program's bin/Debug folder
FileOpen(1, "Runners.txt", OpenMode.Input)

Do While Not EOF(1)
    'Read next two data items, (a name and a time), from the file into the arrays
    'the numElements variable holds the position number, (array subscript), for the item being read
    Input(1, runner(numElements))
    Input(1, time(numElements))

    'increment numElements each time through the loop
    'to move to the next position in the array
    numElements = numElements + 1
Loop

'Close the file before ending the program or reopening the file in another mode
FileClose(1)
```

Data files are usually used with arrays since they do not require all the data to be typed in each time the program is run. An array can also be filled using the InputBox function to get the input from the user if the array is not too large.

```
For position = 0 To 4
    runner(position) = InputBox("Enter a runner's name.", "Runners")
Next position
```

*Averaging the contents of an array:*

```
For position = 0 To numElements - 1
    sum = sum + time(position)
Next position
average = sum / numElements
```

*Changing each entry:*

The following loop changes the time for every runner. (The race-clock was determined to be off by three tenths of a second, so the code shown will add that on to every runner's time.)

```
For position = 0 To numElements - 1
    time(position) = time(position) + 0.3
Next position
```



*Printing an array's contents in a numbered column:*

```
For position = 0 To numElements - 1
    'print the rank and name of each runner
    'rank is not the same as the position in the array
    'the first person in the array has a rank of 1, the 2nd one has a rank of 2, etc.
    outResults.AppendText(position + 1 & "." & runner(position) & vbNewLine)
Next position
```

Here is the output from the code above:

```
1.Speedy
2.BJ
3.Cyclone
4.Angel
5.Lightning
```

### *Parallel Arrays*

The program that follows uses two arrays to hold data that is read from a file. The file contains the names and times of the runners in a race, with one runner's name and time per line and the arrays used are named *runners* and *time*. When reading the file, the first name will be put into the first position of the *runners* array and the first time will be put into the first position in the *time* array. The second name and time are read into the second position of the respective arrays, the 3<sup>rd</sup> name and time into the 3<sup>rd</sup> position, etc. When related data is read into multiple arrays in this way, the arrays are called parallel arrays, since the related data will be in the same corresponding position in all of the arrays.

Speedy,	15.36
BJ,	17.65
Cyclone,	16.88
Angel,	17.18
Lightning,	15.27

Here is what the data file used by the following program looks like:

The program code that follows contains comments that explain what this program will do and the results generated by the program are listed after the program. The entire program is listed on the next page for easier readability.

```

Private Sub btnRaceStats_Click(sender As Object, e As EventArgs) Handles btnRaceStats.Click
    'This sample program illustrates the use of parallel arrays.
    'This program reads a file containing runner names and times into two arrays,
    'and calculates and displays the average time. Then the program prints out
    'a table of all of the runners, their times, and how much each runner's time is
    'above or below the average time.
    'Declare variables ...define the arrays to a size that is big enough
    Dim runner(500) As String, time(500) As Single, avgTime As Single = 0.0
    Dim numElements As Integer = 0, runningtotal As Integer = 0
    Dim aboveBelow As Single = 0.0, frmtStr As String = "{0,-16}{1,10}{2,30}" & vbCrLf

    'open the file to be read (the file should be in the projects bin/debug folder)
    FileOpen(1, "Runners.txt", OpenMode.Input)

    'initialize variables
    numElements = 0          'used for the position within the array
    runningtotal = 0         'used to add up all of the times

    Do While Not EOF(1)
        'Read next data set from the file into the array
        Input(1, runner(numElements))
        Input(1, time(numElements))

        'add current runner's time to runningTotal
        runningtotal = runningtotal + time(numElements)

        'increment numElements each time through the loop
        'to move to the next position in the array
        numElements = numElements + 1
    Loop

    avgTime = runningtotal / numElements    'calculate the average time
    'display the average time
    outResults.AppendText("The average time is: " & FormatNumber(avgTime, 2) & vbCrLf)

    'Print the header info for displaying a table of the runners times
    outResults.AppendText(String.Format(frmtStr, "Name", "time", _
                                        "over(+)/under(-) AVG time"))
    outResults.AppendText("*****" & vbCrLf)

    'loop through the arrays to print the contents
    For position = 0 To numElements - 1
        aboveBelow = time(position) - avgTime 'calculate distance from average
        'output the information to the RichTextBox on the form
        outResults.AppendText(String.Format(frmtStr, runner(position), _
                                            time(position), FormatNumber(aboveBelow, 2)))
    Next position

    FileClose(1) ' Close data file
End Sub

```

The output that would be generated by the previous program is:

```
The average time is:16.47
Name           Time           over(+)/under(-)  AVG time
*****
Speedy          15.36                   -1.11
BJ              17.65                    1.18
Cyclone         16.88                    0.41
Angel           17.18                    0.71
Lightning       15.27                   -1.20
```

The previous sample program uses one dimensional arrays or simple lists, but sometimes two dimensional arrays are a better structure to use to store and use data. For example, the multiplication table that was printed back in section 3.8 could have been stored in a two-dimensional array and then printed. We typically store data in arrays when we need to re-use that information or rearrange it at a later time in our program. One could easily envision a program to help someone learn their multiplication facts where having the facts stored in a table would be useful.

Listed below is code that demonstrates the declaration of a two dimensional array, filling it with data, and printing its contents. Note the Dim statement needs to include the highest subscript for both the number of rows and the number of columns to be in the array.

```
Private Sub btnTwoDimensionalArray_Click(sender As System.Object, e As System.EventArgs) Ha
    Dim multiplicationTable(9, 9) As Integer
    Dim row As Integer = 0, column As Integer = 0, frmStr As String = "{0,5}"

    'fill the array
    For row = 0 To 9
        For column = 0 To 9
            multiplicationTable(row, column) = (row + 1) * (column + 1)
        Next column
    Next row

    'print the contents of the array in table form
    For row = 0 To 9
        For column = 0 To 9
            outResults.AppendText(String.Format(frmStr, multiplicationTable(row, column)))
        Next column
        'move to next row in table
        outResults.AppendText(vbNewLine)
    Next row

End Sub
```

The output that would be generated by the previous program is:

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

## 3.11 Searching Arrays

### Sequential Search

There are many algorithms for searching a list to see if it contains a given entry. The simplest is the sequential search in which the computer begins at the first element of an array and compares each element in sequential order with the data it is searching for. We will discuss and use two types of sequential search, Exhaustive and Match-and-Stop.

### Exhaustive Search

The Exhaustive search begins at the first element and continues sequentially through the entire list, examining every element in the list to determine if it is the one you are looking for or meets the criteria you are searching the list for. Finding the biggest value in a list would be an example of this type of search. The biggest element could be the first element or the last one or any one in between. You have to examine every element to determine which one is the biggest. Counting how many times a particular value occurs in a list would be another example of a problem where an exhaustive search is needed. To tell whether a match was really found we will use a Boolean variable, *found*. This variable is initially set to False, and if a match is found it is changed to True. Suppose we are searching an array of names (perhaps the runners from section 3.10) to count how many runners have a certain name. Again we are assuming the array has been filled with the names, that the numElements variable value is also accurate and that the array and numElements were declared as form-level variables. We will enter the name we are searching for with an input box. Here is the code that demonstrates an exhaustive search to count the number of times a name occurs in the list of runners:

```
Private Sub btnExhaustiveSearch_Click(sender As System.Object, e As System.EventArgs) Handles btnEx
    Dim found As Boolean = False, nameOfRunner As String = ""
    Dim position As Integer = 0, nameCounter As Integer = 0

    nameOfRunner = InputBox("Enter the name to count the occurrences of", "Name")

    For position = 0 To numElements - 1          'searches the entire list
        If nameOfRunner = runner(position) Then 'examine every single entry in the list
            found = True 'we could just rely on the nameCounter value to know if 'found
            nameCounter = nameCounter + 1
        End If
    Next position

    If (Not found) Then                          'prints the results
        outResults.AppendText(nameOfRunner & " was not in the race.")
    Else
        outResults.AppendText("The name " & nameOfRunner & " appeared " & nameCounter & " times.")
    End If
End Sub
```



## Match/Stop search

The Match/Stop search begins at the first element and stops when a match is found or if the end of the list is reached. To tell whether a match was really found we will use a Boolean variable, *found*. This variable is initially set to *False*, and if a match is found it is changed to *True*.

Suppose we are searching an array of names (perhaps the runners from section 3.10) for a particular name. We are assuming that the list of runners has already been read into the String array, *runner* and that the value of *numElements* has been determined. We are also assuming that the array variable, *runner*, and the Integer variable, *numElements*, were declared as form-level variables so that they are accessible in the code for this command button. We will enter the name we are searching for with an input box. In a match and stop search, if the item is found, the search stops at that point. You can also think of a match and stop search as finding the first occurrence of an item in a list. Here is the code that demonstrates a match and stop search to find the first occurrence, (if any), of a given name in the list of runners:

```
Private Sub btnMatchStopSearch_Click(sender As System.Object, e As System.EventArgs) Handles btnMatch!
    Dim found As Boolean = False, nameOfRunner As String = ""
    Dim position As Integer = 0, positionOfRunner As Integer = 0

    nameOfRunner = InputBox("Enter the name you wish to find", "Name")

    'searches until found or end of list
    Do While ((Not found) And (position < numElements))

        'check to see if this position holds the name you are looking for
        If nameOfRunner = runner(position) Then

            found = True      'remember that you found it by setting found to True
            positionOfRunner = position 'remember where runner was found

        End If

        position = position + 1 'increment the array index to the next 'position

    Loop

    If (Not found) Then
        outResults.AppendText(nameOfRunner & " was not in the race.") 'prints the results
    Else
        outResults.AppendText(nameOfRunner & " was runner " & positionOfRunner & " in the array.")
    End If

End Sub
```

## 3.12 Sorting Arrays

### Bubble Sort

There are many algorithms for sorting a list. We will only present one of the simplest ones here. The Bubble Sort makes several passes through a list. On each pass it compares each number with

its neighbor to the right. If a number and its neighbor are out of order, relative to one another, they exchange places. On each pass through the list, the largest number “bubbles up” to its proper place in the list.

#### Bubble Sort Algorithm for an Array containing numElements elements ... **version 1**

1. Make numElements Passes through the array—on each pass:
  - a. Start at the beginning of the array and compare each number with its neighbor to the right and exchange (swap) them if they are out of order.

Consider the list

2      5      4      3      1

On the first pass through the list we start the pass by comparing the first number in the list with the second number, the 2 with the 5. They are not out of order, so we move one element to the right and compare the second number to the third number, the 5 with the 4. They are out of order, so we exchange them, getting:

2      4      5      3      1

Next we compare the third number in the list to the fourth number, the 5 with the 3, exchange them, then the 5 with the 1, exchanging them. After one pass our list looks like

2      4      3      1      5

The 5 is now in place after four comparisons and exchanges as needed. We now start our second pass through the list. We compare the 2 and the 4. They are ok, so we leave them where they are and compare the 4 and the 3. They must be exchanged. Now compare the 4 and the 1 and make one more exchange. We do not really need to compare the 4 and the 5, since we know the 5 is in its proper place from the first pass through the list. This suggests that version 1 of the Bubble Sort Algorithm needs an improvement since there is no need to compare an element that we know is already in the correct location. Here is version 2:

#### Bubble Sort Algorithm for an Array containing numElements elements ... **version 2**

1. Make numElements Passes through the array—on each pass:
  - a. Start at the beginning of the array and compare each number with its neighbor to the right that is not yet in the correct place and exchange (swap) them if they are out of order.

Thus, after pass 2 the list looks like

2      3      1      4      5

In pass 3, we compare the 2 and the 3, leave them, then compare the 3 and the 1 and exchange them. We now have

2      1      3      4      5

In our fourth pass we compare and exchange the 2 and the 1. There is not any reason to pass through the list a fifth time since each pass through the list definitely puts one element in the

correct place. After putting four elements in the correct location, the fifth element must already be in correct location. Our list is now sorted.

1      2      3      4      5

Notice that for a list containing 5 elements we needed 4 passes to get all the numbers in place. On each pass one number “bubbled up” to the correct spot, but on the last pass, our final exchange put two numbers in their correct spot. Thus, for a list of size numElements we need numElements – 1 passes to be sure it is sorted.

The **final version** of the Bubble Sort Algorithm to sort a list of numElements numbers and then print the list is:

1. Make numElements - 1 passes through the list.
  - a. On each pass:  
Start at the beginning of the array and compare each number with its neighbor to the right that is not yet in the correct place and exchange (swap) them if they are out of order.

There are two things we must refine in this algorithm before we can write the code. The first is how many comparisons to make. Notice that on the first pass, with numElements numbers we must make numElements - 1 comparisons. On the second pass we have one number in place, so we make numElements – 2 comparisons. In general, on each pass we make one less comparison than on the previous pass. That can be expressed as (numElements - the pass number) comparisons. When writing the code for the loops controlling the number of passes, remember that in order to repeat a task n times we use a For/Next Loop that goes from 0 to n – 1. To have the code pass through the list (numElements – 1) times, the loop must go from 0 to numElements – 2.

We also need to refine how to exchange two numbers. If we simply wrote

```
list(position) = list(position + 1)
list(position + 1) = list(position)
```

we would end up with two copies of list(position + 1), for when the first line was executed list(position) would get a copy of the contents of list(position + 1) and its old contents would be lost. Thus we need to save list(position) first. We will use a variable called tempNumber, for temporary storage, to do this.

```
tempNumber = list(position)
list(position) = list(position + 1)
list(position + 1 ) = tempNumber
```

\*\*\* Note that when an array of Strings is sorted into ascending order, (alphabetical order), upper case letters come before lower case letters. The letter ‘A’ has an associated value that is less than the value for “B” and all upper case letters have values that are less than the letter ‘a’. And Strings starting with a numerical digit, (ie. A phone number) would come before any Strings beginning with a letter.

If this list: {"cat", "Dog", "apple", "320-3133", "Amazon", "dog", "123"} was sorted into ascending order, it would appear as: {"123", "320-3133", "Amazon", "Dog", "apple", "cat", "dog"}

Next we have a complete coded sample that illustrates the use of a Bubble Sort. The program reads data from a file into an array, sorts the array using the Bubble Sort, and prints the sorted list.

```
Private Sub btnBubbleSort_Click(sender As System.Object, e As System.EventArgs) Handles b
    'Here's the complete code needed to read a list of numbers from a file into an array,
    'sort the array of numbers into ascending order using the Bubble Sort
    'and then print the list of sorted numbers:

    'declare the variables needed
    Dim list(100) As Integer
    Dim pass As Integer = 0, position As Integer = 0
    Dim tempNumber As Integer = 0, numElements As Integer = 0

    'open the file to obtain the data
    FileOpen(1, "myNumbers.txt", OpenMode.Input)

    'fill the array with numElements numbers from the data file
    Do While Not EOF(1)
        Input(1, list(numElements))
        numElements = numElements + 1
    Loop

    'close the file once you are finished reading it
    FileClose(1)

    'use the Bubble Sort to arrange the numbers in the desired order
    'this code sorts the list of Integers into ascending order

    For pass = 0 To numElements - 2                                'keep track of how many passes
        For position = 0 To numElements - 2 - pass                 'keep track of how many comparisons

            If list(position) > list(position + 1) Then
                tempNumber = list(position)                         'exchange values if out of order
                list(position) = list(position + 1)
                list(position + 1) = tempNumber
            End If

        Next position
    Next pass

    'print the sorted list
    For position = 0 To numElements - 1                            'output section
        outResults.AppendText(list(position) & vbNewLine)
    Next position

End Sub
```



When using parallel arrays and sorting one of the arrays into ascending order, you need to be sure to include the code to swap elements of all of the arrays, not just the one you want in order. (i.e. If you change the position of the runner's name in the *runner* array, you must also change the position of the runner's time in the *time* array to ensure that the related data remains in the corresponding positions within both arrays. For each additional array you will need three additional lines of code to swap the elements in that array and an additional temporary variable to hold the data while you are swapping it.

Here is what the Bubble Sort would look like when using the parallel arrays *runner* and *time* and sorting the runner array into alphabetical order:

```
'sort the runner names into alphabetic order
For pass = 0 To numElements - 2
    For position = 0 To numElements - 2 - pass
        If runner(position) > runner(position + 1) Then
            'swap the names for out of order elements
            tempName = runner(position + 1)
            runner(position + 1) = runner(position)
            runner(position) = tempName

            'swap the times, too
            tempTime = time(position + 1)
            time(position + 1) = time(position)
            time(position) = tempTime
        End If
    Next position
Next pass
```

### 3.13 Multiple Forms and Code Modules

#### Multiple Forms

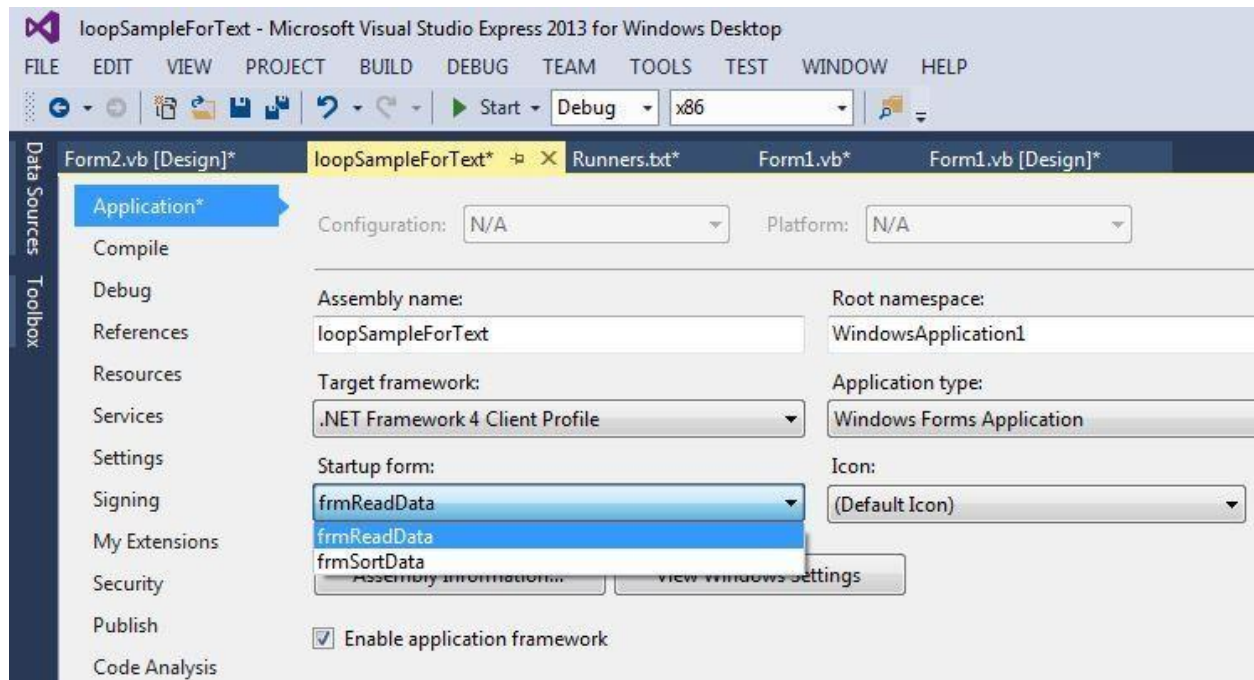
All of our projects so far in this textbook have used a single form. Often, a project with multiple options, such as one that lets you enter data into an array, sort, and search that array, would be better served with separate forms for each of these tasks. It is easy to add forms to a project and to move from form to form. To add a new form to a project, simply use the menu option: Project -> Add Windows Form. A Window will pop up with a place near the bottom to enter a name for the new form. Remember to begin the name with frm, (ie. frmSortData), and click Add. Now you can add buttons and other objects to it as desired and remember to **Save All** before you exit the program so that this form will be saved as part of the current project.

To switch between forms you can have a button on your form that contains the code to hide the one you are on and show the other one. When you want to refer to the form you are currently on with your VB program, you cannot refer to it by name, but must refer to it as Me. The name, Me,

is automatically declared for you and was given a certain meaning as part of the VB language, somewhat similar to the use of `vbNewLine`. The code to switch control from one form to a second form named `frmSortData` is simply these two lines:

```
Me.Hide()  
frmSortData.Show()
```

Sometimes when you create a project with multiple forms you may decide that you'd like the second form you added to the project to be the first form shown to the user, also known as the startup form. You can change the startup form for your project using the menu option: `Project->yourProjectName Properties`. This will bring up a window where several settings can be changed. Within the `Application` tab there is a drop-down menu choice labeled "Startup form:" where you can select the form you want your project to start with from the list shown. Here is a screen shot showing the Startup form pane with the drop down menu arrow:



## Code Module

You may wish to access the same variable(s) on more than one form. Recall that variables declared with the `Dim` statement within a subroutine are accessible only to that subroutine while those declared at the top of a code page are accessible to all the events on that form. When a project involves multiple forms from which we would like to be able to access the data, the variables we wish to access from multiple forms need to be declared as **Public** to allow access to them from any form within your project. This requires the use of a code **Module**. You need to explicitly add the Module to your project where you will then put the declarations of **Public** variables (all variables you want to be able to access from anywhere within your project). If you

have a project with multiple forms that has buttons on two or more forms that need access to the same arrays, declaring the arrays as form level variables or subroutine level variables would not allow access to that data from multiple forms. The array variables and the associated numElements variable would need to be declared as Public in a code module. You can add a Module to your project using the menu option: Project -> Add Module and then add variable declarations to it. For example, if we wished to have, the *runner* and *time* arrays used in a previous example accessible to multiple forms, the code module would contain the lines:

```
Module Module1
    Public runner(75) As String, time(75) As Single, numElements As Integer = 0
End Module
```

The declaration of variables as **Public** takes the place of using a **Dim** statement for those variables.

## Form Load

Sometimes you may want to initialize variables associated with a form when it is loaded instead of having a button that includes code to initialize them. There is a special subroutine called “Form\_Load” that can be used to place code that you want to run each time the form is loaded. This can be used to simply initialize variables or do something much more involved such as read data into an array from a file. To get to the Form\_Load subroutine, just double click any blank area on the form and the code headers will appear like they do when you double click a command button, allowing you to enter the code you want to have run each time the form is loaded. The Form\_Load is not a click event; it is just executed each time the form is loaded. Your code would be entered between the Private Sub and End Sub lines, as is done with command buttons. Here is a sample that shows a form load event for reading a data file into the runner and time arrays declared as Public in the sample Module code listed above:

```
Private Sub frmRaceData_Load(sender As System.Object, e As System.EventArgs) Handles MyBase.Load
    'Prepare the file to be read (Note: text file should be in the bin/debug folder for this project)
    FileOpen(1, "Runners.txt", OpenMode.Input)
    'initialize numElements to zero, to be used for position in the array
    numElements = 0
    Do While Not EOF(1)
        'Read next data set from the file into the array
        Input(1, runner(numElements))
        Input(1, time(numElements))
        'increment numElements to move to the next position in the array
        numElements = numElements + 1
    Loop
    ' Close file
    FileClose(1)
End Sub
```

### 3.14 Writing and Using Your Own VB Functions

In section 3.5 you were introduced to several numeric and string functions that are part of the VB programming language. Sometimes it would be useful to have a function available in your program that does something not found in any of the built-in functions. In VB, you can create your own functions to do what you want them to do. A function is just a conveniently packaged program that you give some data to, and it performs some desired calculation or manipulation of the data, and returns a result. Consider the built-in function that will calculate the square root of a number. To use the square root function, you have to pass it a number and it will do the calculation to compute and then return the square root of that number. Typically, you use a function as part of an assignment statement that calls a function and assigns the value that it returns to a variable. (ie. `myFavoriteSquareRoot = sqr(2)` ) The square root function only needs to be passed one piece of data in order to do its job. Other functions may require multiple data items to calculate the desired result. Several things need to be determined before you write your own function. Let's assume we would like to write a simple function to calculate a person's paycheck based on the number of hours worked and the hourly rate of pay. As you write functions there are a few things that need to be determined before you can write the VB code:

1. **Function Name:** What is the name of your function? We will use the camelback scheme when naming functions. For our example let's use 'calculatePay()' as the function name.
2. **Parameter List:** What parameters need to be passed to the function? What information will your function need to be passed in order to do the desired calculation(s)? (parameter1 As Datatype1, parameter2 As Datatype2, ...) For our *calculatePay* function we have two parameters, hours and payRate, both of Datatype, Single.
3. **Return Type:** What type of data will your function return? What type of result do you wish to get from your function? (A String, Boolean, Single, Integer ...?) For our *calculatePay* function, we would like it to return a result of type Single.
4. **Calculations:** How will you process the input data to get the desired result? Are you using a formula or an expression of your own design? For our *calculatePay* function we will calculate the pay by simply multiplying the two parameters. (ie.result = hours \* pay)

The information just gathered needs to be put into a prescribed pattern. In VB, the pattern or template for every function looks like this:

```
Function <functionName> (<parameter list>) As <return value data type>
    Dim result As <return value data type>
    Dim <any other variables needed for your function>
    < code needed to calculate the result >
    < this will include an assignment statement involving result>
    < (ie. result = hours * payRate ) >

    return result
End Function
```

Using the decisions listed for designing the *calculatePay* function, here is the code for the function:

```
Function calculatePay(hours As Single, payRate As Single) As Single
    Dim result As Single

    result = hours * payRate

    Return result
End Function
```

In order to be able to use your functions on any form of your VB project, the function definition needs to be put into a code Module. Add a Module to your project before you write the code for your function, and then write the code in the module as shown below:

```
Module MyFunctionModule
    Function calculatePay(hours As Single, payRate As Single) As Single
        Dim result As Single

        'do the calculation to determine pay and assign it to 'result'
        result = hours * payRate

        Return result
    End Function
End Module
```

## Using Your Functions

Once you have written your function and added it to a module, you can “call it”(use it), from within your code on any of the forms of your VB project by simply referring to the function name and passing information to it with its parameters in parentheses. Here is sample code for a button that would get information from the user from text boxes and then use the *calculatePay* function to determine the amount of pay.

```

Private Sub btnCalculatePay_Click(sender As System.Object, e As System.EventArgs) Handles btnCalculat
    'declare variables
    Dim myPay As Single, myHours As Single, myHourlyPayRate As Single

    'get the user input
    myHours = txtHours.Text
    myHourlyPayRate = txtRate.Text

    'call your function to calculate the pay, passing the user input as parameters to your function
    myPay = calculatePay(myHours, myHourlyPayRate)

    outResults.AppendText("You earned " & FormatCurrency(myPay) &
        " for the " & myHours & " hours you worked.")
End Sub

```

The sample function used above has a very simple calculation and only two input parameters. You can create much more complex functions that do extensive calculations, use loops, if statements, file input, etc. Remember that a function is just a convenient way to do a task that you may want to do often, but instead of needing to write the code each time, you can just call your function to do the desired task.

### 3.15 Conclusion

This is only a brief introduction to programming in Visual Basic. There are many additional features that we have not covered. However, by now you should have sufficient practice writing VB programs to see the power of such a high-level language for programming a variety of applications. With this basic grasp of the programming process and the essential elements of VB you should now be able to design your own short programs. Should you need objects or techniques that we have not covered, you now have the vocabulary to use and understand materials written to help one learn more about the VB language.

Here is a link to some online resources provided for Visual Basic by Microsoft:

<http://msdn.microsoft.com/en-us/library/2x7hlhfk.aspx>

Though we are moving into applications, we will not be leaving Visual Basic far behind. A variant of Visual Basic called Visual Basic for Applications (VBA) is the underlying programming language for all the applications in Microsoft Office. When we explore the spreadsheet application, Excel, we will use VBA to tailor spreadsheets to our own needs and when we explore the database management system, Access, we will use VB programs to connect to a database file to retrieve information from it. There is also a section in the Appendix that illustrates how to use VBA within an Access database to write programs that interact with the database. Visual Basic will give us the power to go beyond the built-in functions in these applications.

# *Chapter 4*

## *Databases: An Introduction to Access 2016*

### *4.1 Basic Database Principles and Features*

Before databases existed, companies maintained their data in disjoint sets of files and wrote application programs to access/update these files. This led to separation and isolation of data, duplication of data, inconsistencies, incompatibilities, etc. The database and database management system were created to separate the data from the application programs and to provide controlled access to the data.

A database is an organized collection of logically related data (and a description of this data), designed to meet the needs of an organization.

A DBMS (Database Management System) is a software system that enables users to define, create and maintain the database and provides controlled access to the data.

Properties of a full-featured DBMS:

1. Concurrent access
2. Data integrity/consistency support
3. Data independence (logical/physical)
4. Transaction support
5. Data view definition
6. Security
7. Query ability
8. Backup and Recovery
9. A user-accessible system catalog (data metadata)

A Relational Database Management System (RDBMS) uses the relational data model to accomplish these tasks. In the relational model, the data is logically related by being broken down into multiple tables, each with a unique identifier called a primary key which is used to establish a link (or relation), between one table and another.

Microsoft Access is a RDBMS and we will use it as the database platform in this course to introduce you to designing and using databases.

#### *Database Keys*

A database table looks very much like an excel spreadsheet. We call the rows “records”, the columns “fields” and the spreadsheet itself a “table”.



Each table needs one *primary key* which can be a single field or a combination of fields to act as a way to uniquely identify each record. No two records can share the same values for their primary key field(s). Common primary keys are ID numbers like Social Security Numbers, VINs, Banner IDs, ISBNs, etc.

When a table contains a value that matches up with a different table's primary key, that field acts as a *foreign key* to establish a link between the tables. An example would be a registration table that has a student id number which points to the student table.

A company's database typically consists of a set of *tables* holding interrelated data. Each table holds data for one area of the business, such as products, employees, customers, vendors, etc. (The screen shots in this chapter were taken from the Gargoyles database found on the N: drive in the AccessExamples folder. You may want to open that database within Access now and try things out as you read.) Consider the following table holding information about the employees of Gargoyle Computing:

Employees							
Employee Number	FirstName	LastName	Department	Date of Hire	Full Time	Salary	Bonus
957-787	Allison	Simons	Sales	12-Apr-99	<input type="checkbox"/>	\$37,575	\$150
663-234	Brian	Cutter	Sales	21-Mar-00	<input checked="" type="checkbox"/>	\$26,133	\$275
523-456	Catherine	Donovan	Sales	30-May-98	<input checked="" type="checkbox"/>	\$26,133	\$300
332-345	Crystal	Jameson	Sales	03-Mar-98	<input checked="" type="checkbox"/>	\$26,133	\$400
224-987	Mike	Lorenz	Sales	15-Oct-98	<input checked="" type="checkbox"/>	\$40,248	\$350
151-009	Gail	Norton	Sales	21-Sep-98	<input checked="" type="checkbox"/>	\$26,133	\$200
123-543	Mikey	Lorenz	Sales	15-Oct-98	<input checked="" type="checkbox"/>	\$67,514	\$350

The entire table is named Employees. The table is divided into rows, called *records*. Each row in this table contains the data for one employee. Each record is divided into *fields*. The fields form the columns of the table, each headed by a *field name*, such as LastName or Department. The names represent the type of information that will be stored in that field for each record.

One field in the table must have a unique entry for each record to assist in maintaining data integrity. There must be some way to find each record for quick retrieval. Such a field is called a *key field* and one such key field is designated as the *primary key* for each table. In the table above, the Employee Number is the primary key.

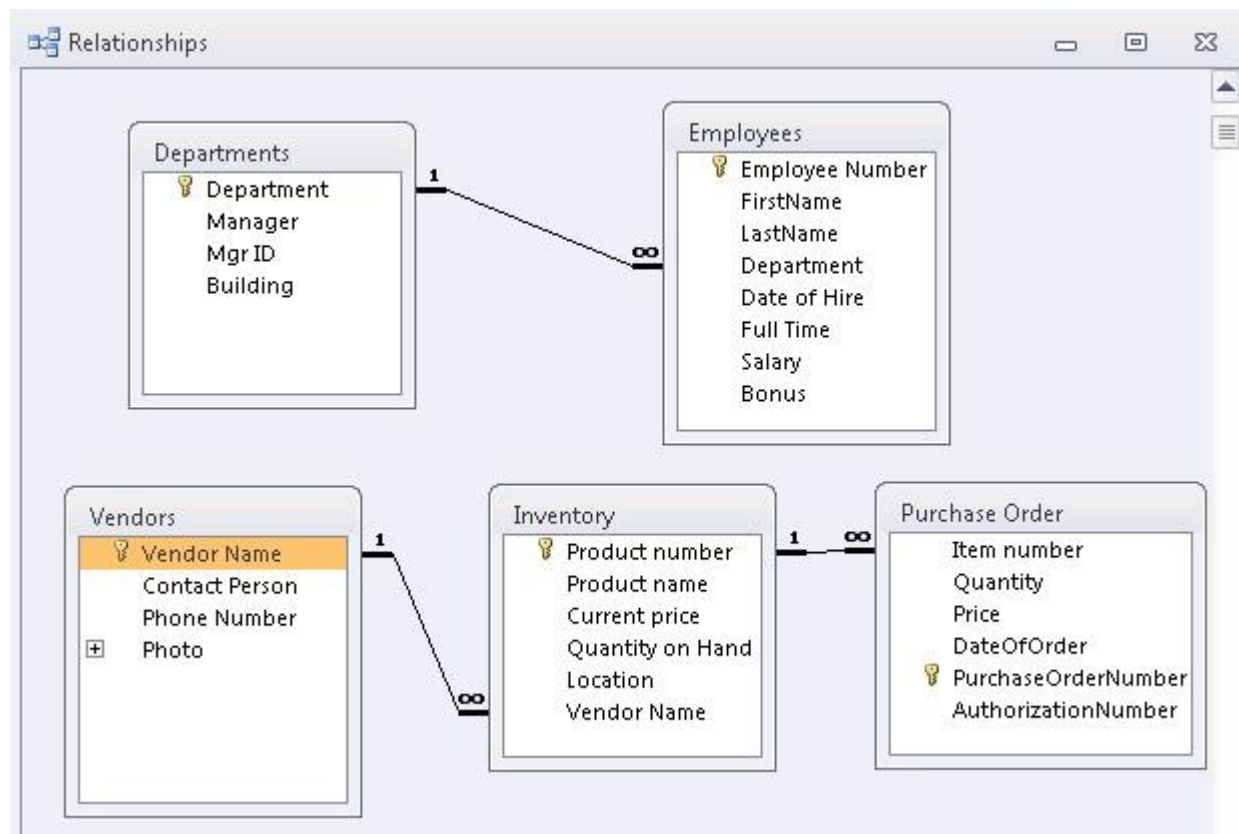
## 4.2 Designing a Database

A good relational database requires a lot of up-front work designing the right table structure before the database application is even started. It is very important to know what kind of application you are building and to gather information from the users of the system to determine exactly what data they require in the application. It is unlikely that you would want all of the information in a single large table. Most databases consist of several tables, with some tables linked to others through common key fields. Once you determine what information you would



like to store, you should design a database model containing the tables, fields and links between the tables before creating the structure with the database software.

For example, Gargoyle Computing may wish to store information on its employees, departments, vendors of supplies, purchase orders, and inventory. These tables might be linked as follows:



Notice that not all tables are linked. Employees and Departments share common data, but do not share data with the other tables. Typically links between tables are one-many. A department in a company has many employees but an employee works in only one department at a time. Some tables, while not linked directly, are linked through another table. Vendors is not directly linked to Purchase Orders, but both are linked to Inventory, so we could combine data from the Vendors and Purchase Order tables by following the link through Inventory.

The fields that are linked need not have the same name, but they do need to contain data of the same type with some of the same entries. For example, the Item number field in the Purchase Order table is linked to Product number field in the Inventory table. These fields do not have the same name but both contain the same possible entries. Department is a key field for the Departments table; each entry is unique, since there would not be two departments with the same name in a company. However, Department is not a key field for the Employees table, as there would likely be more than one employee in any given department. When linking tables, one of the fields linked must be a key field, but they need not both be. The link between Employees and Departments is what we call a one-to-many link, because one department name is linked to the records of several employees. One-to-many links are denoted with a one way arrow:



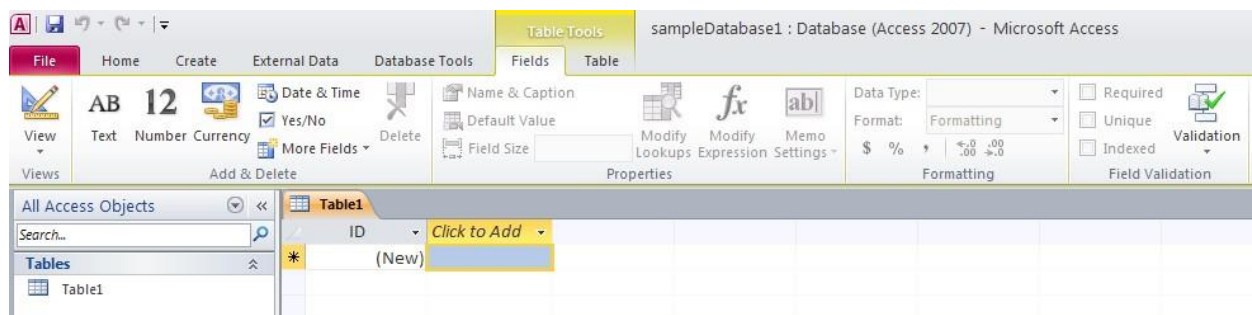
Links may be one-to-many or one-to one. Links cannot be many-to-many. The relationships between tables will be discussed in greater detail later in this chapter.

Once you have determined what tables to build, what the key will be for each table, and how the tables will be linked you then need to determine exactly what fields are necessary for each table, and what properties you wish those fields to have. It is best to design your database on paper, making sure each table has the information necessary before creating the database on the computer.

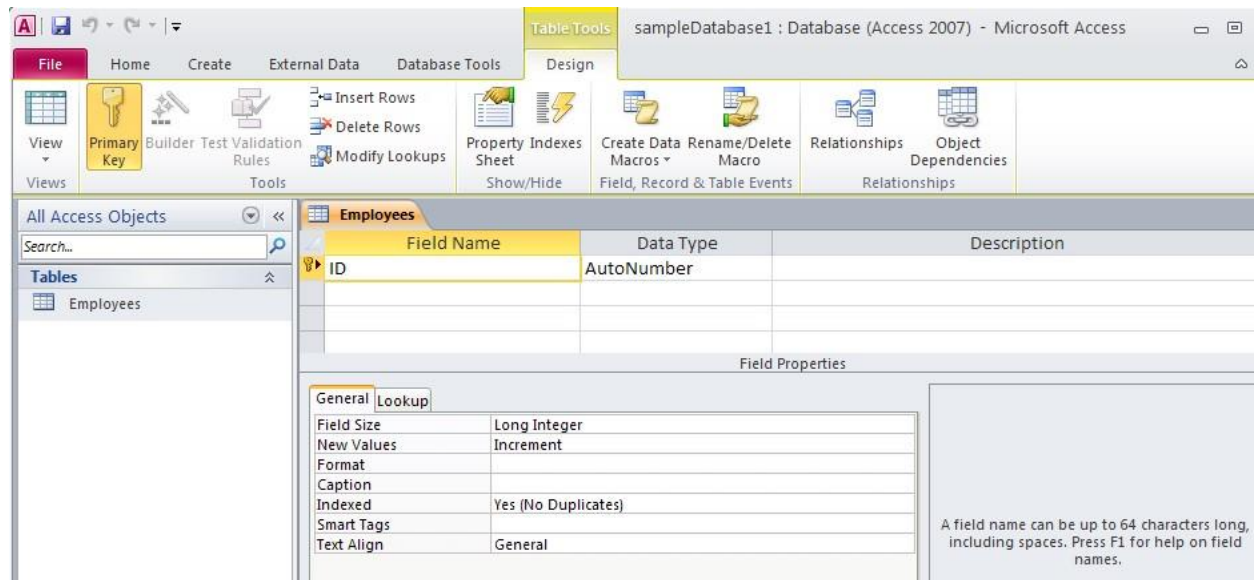
### 4.3 *Creating Tables in Access 2016*

#### *Designing a Table*

Start a new database: Open Access and choose Blank Database, give it a name and click Create. If Access is already started you can start creating a new database by choose File -- New and then type a name for this new database file, choose its storage location and click on Create. The introductory screen shows the Home tab ribbon and the screen is displaying the DataSheet view of an empty table, ready for the user to start adding data. The introductory screen is listed below.



Rather than entering data into a table called Table1 using the default field names of Field1, Field2, etc., it is helpful to immediately switch to the Design View. To switch to design view, from the Home or Datasheet ribbon, within the Views group, click on the pencil and ruler icon or choose View, and then select Design View. If you are just starting creating a new database, you will be asked to give a name for your first table. In Design View you make important design choices for each field in the table such as appropriate names, data types, descriptions, and any special characteristics before entering any data. Once you choose the data type for a field, the properties window at the bottom of the screen displays the characteristics that can be set for that data type and allows you to make the choices you want. The initial Design View for an empty table looks like the following:



Field names are single strings that begin with a character and contain up to 64 characters. The field data type can be one of the following:

**Short Text** -- used to store short strings of characters (up to 255). May include letters, numbers, punctuation, spaces and symbols. Default size is 50, but you can change this in the properties window as you create the field. You can also set a predefined or custom format to determine how the field will be displayed, such as (@@@)@@@-@@@@ which will cause the phone number 3203425621 to be displayed as (320)342-5621.

**Long Text** -- used for longer strings with more than 255 characters

**Number** -- used for numeric data used in calculations. For numbers you will need to choose the type of number (long or short integer, single or double for real numbers involving decimals) from a pull down menu in the properties window. You can specify the number of decimal places for real numbers. All numbers will have a default value of 0 unless you set the default to something else.

**Date/Time** -- you can choose from various formatting styles of date, time or both. The options available are dependent on Regional Options in the Windows Control panel that can be changed to reflect the preference for a region or country. Our systems at CSB/SJU have the default Regional settings set to English(United States) and using that option May 1<sup>st</sup>, 2008 may be displayed as 01-May-08 if the Medium Date format is used and as 5/1/2008 if the Short Date format is used. There are many different format choices.

**Currency** -- stores monetary values. You can specify the format, decimal places, and default.

**AutoNumber** -- this field automatically numbers your table records, with the first as 1, second as 2, etc.

Yes/No – similar to Boolean data type in Visual Basic, this type produces a check box in the record that stores yes (checked) or no.

There are several more data types available that we will not be using in this course.

There are other properties that you might wish to set for some of your fields. Among the most useful of these are Input Mask, Validation Rule, and Required. An input mask sets the pattern for data to be entered into a field. You can choose from common masks by clicking on this property and then on the menu button that appears. This will start a Mask Wizard that will help you determine a suitable mask for your data. A Validation Rule lets you specify requirements for a valid entry for a given field. For example, for the department field in the Employees table, you might limit entries to valid departments by entering In(“Sales”, “Production”, “Management”, “Design”). Under Validation Text you would give the message to be displayed when someone enters data that does not fit the validation rule, such as “Not a valid department.” The “Required” property lets you determine whether a user must make an entry in this field for each record. A completed table design might look like the following:

Employees		
Field Name	Data Type	Description (Optional)
Employee Number	Short Text	
FirstName	Short Text	
LastName	Short Text	
Department	Short Text	
Date of Hire	Date/Time	
Full Time	Yes/No	
Salary	Number	
Bonus	Number	

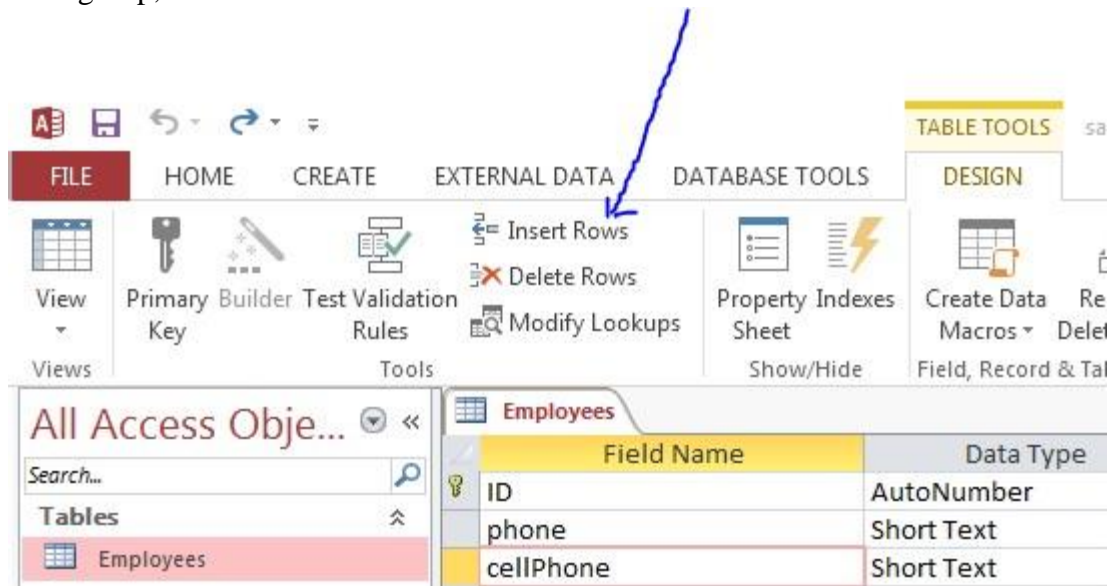
  

Field Properties	
General	Lookup
Field Size	15
Format	
Input Mask	
Caption	
Default Value	
Validation Rule	
Validation Text	
Required	No
Allow Zero Length	Yes
Indexed	Yes (Duplicates OK)
Unicode Compression	Yes
IME Mode	No Control
IME Sentence Mode	None
Text Align	General

A field name can be up to 64 characters long, including spaces. Press F1 for help on field names.

Before saving your table, you must designate one field as the primary key. Position the pointer in the leftmost column of the chosen field and then click on the key icon in the Tools group on the Design ribbon. The key field will be designated by a small key in the leftmost column, as you see next to the first field in the table design above.

If you later wish to alter the structure of a table, you can return to the design mode at any time, either from the initial window, or by clicking on the leftmost icon on the toolbar. To add a new field, click on the row that you wish to follow the new field, then on the Design ribbon, in the Tools group, click on the Insert Rows icon.



To delete a field, click on the field you wish to delete and then on the Delete Rows icon. To rearrange fields, click on the leftmost column of a field you wish to move and then drag it to the desired location.

### Adding and Editing Data

To add data to a field, you must switch from the Design view to the Datasheet view. You can move between these two views by selecting the Home ribbon, and clicking on the icon at the top of the Views group. The icon toggles between images of a pencil and ruler, (design view) and a picture of a table, (datasheet view.) The Datasheet view shows the table's fields as columns. Each row represents a single record. You can add data simply by typing in one row at a time. If a column is not wide enough for a data entry, you can resize it by moving the cursor to the border of that column in the field name section, clicking and dragging the border to the appropriate width. You can also set a column to a particular width by right-clicking the name of the field and then choosing Column Width and entering a width.

If you have provided formats for the data, you need only enter the data itself. For example, a telephone number formatted with (@@ @)@@@-@@@@ will appear in the datasheet as (320)363-2000 but you need only type in 3203632000. The formatting will be added automatically. Similarly, for currency you need not enter the \$. The figure below shows a portion of the same Employees table in Datasheet view:

Employees									
	Employee Numb ▾	FirstName ▾	LastName ▾	Department ▾	Date of Hire ▾	Full Time ▾	Salary ▾	Bonus ▾	Add New Field
+	123--32	Harold	Oslo	Accounting	23-May-98	<input checked="" type="checkbox"/>	\$42,000	\$300	
+	123-543	Tobias	Arnold	Sales	09-May-98	<input type="checkbox"/>	\$39,500	\$200	
+	123--56	Lynn	O'Connel	Production	17-Jul-00	<input checked="" type="checkbox"/>	\$36,000	\$275	
+	142-457	John	Amundson	Managers	12-Jun-95	<input checked="" type="checkbox"/>	\$62,575	\$500	
+	151-009	Gail	Norton	Personnel	21-Sep-98	<input checked="" type="checkbox"/>	\$41,000	\$200	
+	156-098	Patty	McCloud	Purchasing	11-Feb-99	<input checked="" type="checkbox"/>	\$32,000	\$150	
+	224-987	Mike	Lorenz	Sales	15-Oct-98	<input checked="" type="checkbox"/>	\$40,000	\$350	
+	234-944	Jane	Palmer	Design	23-Apr-98	<input checked="" type="checkbox"/>	\$55,600	\$375	
+	253-754	Avery	Winston	Managers	15-Jun-95	<input type="checkbox"/>	\$63,500	\$200	

You can edit data in the datasheet view simply by clicking on the cell you wish to change and entering the new data. In a large database, you can rapidly locate a particular cell using the Find icon. Click on any cell in the field you wish to search in, and from the Home ribbon, in the Find group, click on the Find icon. You can now enter your search criterion. The computer will search the database, highlighting the first record it finds that matches your criterion.

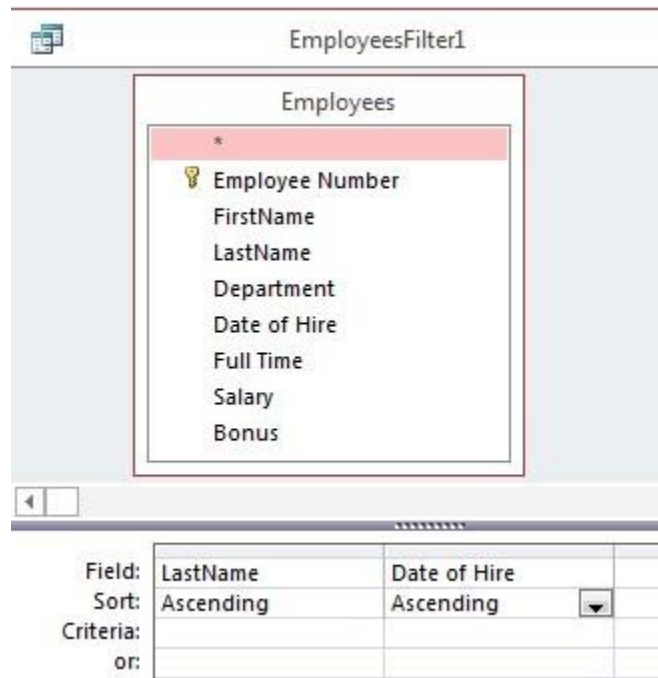
You can delete a record by clicking in the leftmost column to highlight that record, then choosing Delete from the Records group on the Home ribbon, or by right-clicking and choosing Delete Record. You can add new records by moving to the first blank row, and entering the data for the new record. You can also move the cursor to the end of the table and supply an empty row for the new record, if one does not already exist, by using the Home ribbon and Records group and clicking on the New Record icon.

## Sorting

You can sort a table on a single field in ascending or descending order by clicking anywhere in the column you wish to sort by and then clicking on the appropriate sort icon in the Sort & Filter group on the Home ribbon.

To sort on more than one field, click on the arrow next to “Advanced” from the Sort & Filter group on the Home ribbon in Datasheet view, then select “Advanced Filter/Sort. A Filter dialog window will appear. Choose the primary field you wish to sort on in the first column of the Field row from the list of fields provided in the drop-down menu and then choose the type of sort, ascending or descending, from the Sort drop-down menu. Click on the next column over and choose the secondary field and sort type. To perform the sort, click on the Apply Filter/Sort icon under the Advanced menu in the Sort & Filter group on the Home ribbon and the filter will be applied.





To undo a sort, you can click the undo icon in the upper left corner of the window, just above the File tab or you can clear all sorts by clicking on *Remove Sort* in the Sort & Filter group in the Home ribbon.

## 4.4 Queries

A query is a function that selects certain fields and records of a table according to certain criteria and is used to select information to be displayed in a new table or used in a report. Select queries are the most common query form. They produce a new table containing a subset of information from your database, limiting that information so that only certain fields are displayed and only those records that meet the criteria you set. Select queries are also used to summarize or do simple calculations on the data in your database. To determine what information to display, the queries may employ several different methods in setting up the criteria, including using Boolean conditions, concatenation, and calculations.

### Conditional Queries

Suppose we wished to receive a listing of the names and salaries of all full time managers of Gargoyle Computing. We would create a select query to do this. To create such a query, click on the Create tab and then click the Query design icon in the “Queries” group. You will be asked to choose which table(s) you wish to include in your query and a Query design window will appear. Add the relevant table(s), in this case just the Employees table, and then click on “close”. Now choose the fields you wish in your query. You will need to include any fields you wish to have displayed, as well as any fields that are part of the criteria for choosing records. Choose the field names from the drop-down menus provided.

For each field, you can decide whether you want your outcome sorted on that field, whether you want that field to be displayed in the query datasheet, and what criteria data in that field must meet. Criteria are generally given as logical expressions. Here are several sample criteria:

Criteria	Meaning
“Josh”	Field value must equal Josh literally (including case!)
= 4.0	Field value must equal 4.0 (number with decimals – omit decimal for integer comparison)
Yes	Field must equal Yes. For Yes/No (Boolean) Fields
Between 1 and 30	Field value in a specific range – the end points are included
> 45	Field value must be greater than 45; you can use >, <, >=, <=, <>
“Josh” or “Ben”	Field must equal one of the two values
In(“Foo”, “Bar”, Glah”)	Field value is one from a specified list (shorthand for many Or’s)
Not “Josh”	Field value is anything EXCEPT Josh
Like “M*”	Field value is anything that starts with M
Like “*es”	Field value is anything ending in es
> #1-Feb-12#	Field must have a date after Feb 1 <sup>st</sup> , 2012. Dates use “#” signs as delimiters.
[What Department?]	Parameter Query that asks user for the Department Name

Here is the query design for names and salaries of full time managers of Gargoyle Computing:

The screenshot shows the 'Fulltime Management Employees' query design view. On the left, the 'Employees' table is listed with fields: Employee Number (primary key), FirstName, LastName, Department, Date of Hire, Full Time, Salary, and Bonus. The design grid at the bottom shows the following configuration:

	LastName	FirstName	Salary	Department	Full Time
Field:	LastName	FirstName	Salary	Department	Full Time
Table:	Employees	Employees	Employees	Employees	Employees
Sort:					
Show:	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Criteria:				"Management"	Yes
or:					

When finished designing a query, you can run it by clicking on the Run icon, (the red exclamation point), in the Results group of the Design ribbon. If the results are not what you expected or desired, you can make changes to the design by choosing the design view and then making any adjustments needed and then you can toggle between design view and running the query until you are satisfied the query works as you desire. Save the query by clicking on the



save icon above the File button and give it a meaningful name. It will now appear on the Queries section of the Navigation pane. To run the query, double click on the query name in the Navigation pane, or select open from the menu when you right-click the mouse. You will see a datasheet with the results:

Fulltime Management Employees		
LastName	FirstName	Salary
Miller	Lois	\$82,703
Winkler	Nadine	\$81,171
Bonds	Melvin	\$83,597
Wangenleit	William	\$96,870
Satura	Luke	\$76,449
Tom	Walter	\$90,617
Benson	Frances	\$84,873
*		\$0

### Calculated Fields within a Query

Criteria in a select query can do more than choose certain records using the value of a single field. You can combine fields when designing a query to form new fields. You can create a new Calculated Field by doing arithmetic or concatenation operations using the data from existing fields. We do this by putting the name of the new calculated field in the Field row of the query design followed by a colon ':' and then following the colon with the arithmetic or concatenation expression that calculates the desired value of this new field using the values from existing fields. For example, assuming the Employees table has separate fields for the first and last name, a calculated field for the full name of an employee can be created by concatenating those two fields as indicated on the next line. \*\*Note the field names must be put in square brackets.

Full\_Name:[LastName] & " " & [FirstName]

A query that used the Full\_Name calculated field would present the data as seen below:

Winkler, Nadine  
Wangenleit, William  
etc.

As another example, suppose we wanted to create a new calculated field called Adjusted\_Salary that would display the sum of the Salary and Bonus fields in the Employees database. We could generate a table that would display both this new calculated field, "Adjusted\_Salary" and the Full\_Name calculated field with the following query:

Field:	Full_Name: [LastName] & ", " & [FirstName]	Adjusted_Salary: [Salary]+[Bonus]
Table:		
Sort:	Descending	
Show:	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Criteria:		
or:		

This would produce a new salary for each employee. A portion of the results are shown below:

Salary with Bonus (Calculated Field)	
Full_Name	Adjusted_Salary
Winston, Avery	\$63,700.00
Winkler, Nadine	\$64,080.00
Wangenleit, Williar	\$76,400.00
Thomas, Walter	\$71,480.00
Thomas, Robert	\$56,400.00
Simons, Allison	\$37,725.00

### Using Built-in Functions within a Query

In creating a select query, you can also use a variety of built-in functions that calculate and display some summary information about a field or you can create your own calculated fields that do simple calculations. For an example of using the built-in functions, the following query reports the sum, average, maximum, and minimum salaries for Gargoyle Computing:

Salary Data (uses several aggregate functions)				
<div>Employees</div> <ul style="list-style-type: none"> <li>Employee Number</li> <li>FirstName</li> <li>LastName</li> <li>Department</li> <li>Date of Hire</li> <li>Full Time</li> <li>Salary</li> <li>Bonus</li> </ul>				
Field:	Salary	Salary	Salary	Salary
Table:	Employees	Employees	Employees	Employees
Total:	Sum	Avg	Max	Min
Sort:				
Show:	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Criteria:				
or:				

Notice that there is a new row, Total, on the design form of this query. This row is used for the built-in functions and can be added to your query sheet by clicking on the Totals icon in the Show/Hide group of the Query Tools ... Design ribbon. You can then select the function you want from the pull down menu in this row. The output from this query would be:

Salary Data (uses several aggregate functions)			
SumOfSalary	AvgOfSalary	MaxOfSalary	MinOfSalary
\$1,926,450.00	\$45,867.86	\$75,900.00	\$21,500.00

There are also several built-in functions for using dates as part of a query. Within the data-base program and in many other applications, dates are represented internally as a decimal number that represents the number of days, with a zero or start-date of midnight, December 31st, 1899. Each day adds one to the date value. (ie. Noon on January 2<sup>nd</sup>, 1900 would have a date value of 2.5, two and one half days since midnight of Dec. 31, 1899.) The integer portion of the date is the number of days and the fractional part is the time of the day. Here is another example: Dec. 14, 2012 at 1:30 pm has a date value of 41257.56, or when formatted it looks like this: 12/14/2012 13:30.

The built-in date functions described below use the underlying numeric representation to calculate and display the current date, or other information about the date, such as the year, month, or day-of-the-month.

Built-in Date Functions	What this function does
Now()	Returns the date-value for current date and time (ie. 41257.56)
Year(someDate)	Returns the four-digit integer value for the year of someDate (ie. 2013)
Month(someDate)	Returns the integer number of the month of someDate, (January = 1)
Day(someDate)	Returns the integer day-of-the-month of someDate, (1 to 31)

One can use the date functions within a query to choose the desired records. For example, if you wanted to find the names of employees hired in the first half of 1999, you could use the Year and Month functions as part of two calculated fields to make your selection. A sample query design is shown below. Note that the names of the fields are enclosed in square brackets within the parentheses for the functions:

Field:	LastName	FirstName	Salary	Year Hired: Year([Date of Hire])	Month Hired: Month([Date of Hire])
Table:	Employees	Employees	Employees		
Sort:					
Show:	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Criteria:				1999	<7
or:					

Sample output from the query above:

LastName ▾	FirstName ▾	Salary ▾	Year Hired ▾	Month Hired ▾
McLoudi	Patty	\$40,841	1999	2
Jagadinsky	Marty	\$50,668	1999	4
Gohl	Tom	\$53,604	1999	3
*		\$0		

You can also choose dates using expressions such as:

- > (Now() - 30)      which would refer to any date within the past thirty days
- < (Now() - 21)      which would refer to dates more than three weeks ago

### Formatting Fields within a Query

When creating a query, you can change the format of a field. While creating the query, go to *Design View*. In the query design area, click the name of the field you would like to format. On the right of your screen, there should be a Property Sheet pane where you can make your formatting choices. If the Property Sheet pane is not visible, click the Property Sheet icon in the Show/Hide tab on the main menu ribbon, (far right side.) You can choose from a variety of formats. If the field you are formatting is a number, you can change how many decimal places to display by choosing Fixed from the Format menu, and then also changing the entry for Decimal Places.

## 4.5 Advanced Queries.

### Parameter Queries

Parameter queries allow the designer to substitute an input box for a criterion. This allows the user to specify the criterion each time the query is run. A parameter query is more general than a select query, and a good choice if you know you will want similar subsets of data, where the criterion may vary. For example, suppose we wanted the names of employees in a given department. We could write four separate queries, one for management, one for production, one for design, and one for sales. A better way is to write a parameter query with an input box for the name of the department. To get a list of employees in a given department, one would first enter the name of the department, then run the query. Such a query for the table Employees follows:

Field:	LastName ▾	FirstName	Department
Table:	Employees	Employees	Employees
Sort:			
Show:	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Criteria:			[Enter Department Name]
or:			

Note that the criterion must be in square brackets. When executed, an input box appears:



Enter Parameter Value

Enter Department Name

Sales

OK Cancel

Clicking on ok gives the names of all employees in the department entered:

Employees by Department(parameter query)		
LastName	FirstName	Department
Arnold	Tobias	Sales
Lorenz	Mike	Sales
Jameson	Crystal	Sales
Donovan	Catherine	Sales
Cutter	Brian	Sales
Simons	Allison	Sales
*		

### Crosstab Queries

Crosstab queries produce a table that summarizes information across two fields. Suppose we wanted to compare the average salaries for full and part time employees across departments. We would like a chart that shows the departments as columns, with full time and part time as two rows and the average salaries in the body of the chart. To create a crosstab query, begin a new query in design mode and then choose Crosstab from the Query Type group on the Query Tools Design ribbon. Choose Group By in the Totals row for the fields that form the rows and columns and Avg in the Totals row for the field that provides the data for the body of the table. In the Crosstab row indicate which fields make up the row headings, column headings, and values.

A crosstab query to show the average salary for full and part time Employees by department would be:

Field:	Department	Full Time	Salary
Table:	Employees	Employees	Employees
Total:	Group By	Group By	Avg
Crosstab:	Column Heading	Row Heading	Value
Sort:			
Criteria:			
or:			

The output of this query is the following chart:

Average_Salary_by_Dept_(crosstab query)							
Full Time	Accounting	Design	Managers	Personnel	Production	Purchasing	Sales
<input checked="" type="checkbox"/>	\$41,857.14	\$58,550.00	\$66,221.88	\$48,833.33	\$41,875.00	\$32,750.00	\$39,375.00
<input type="checkbox"/>	\$32,000.00	\$56,250.00	\$63,500.00		\$23,833.33		\$38,537.50

## Action Queries

Select queries create a new table meeting the criteria and format given in the query. Action queries do not create a new table but change the data in the original table. Action queries are extremely powerful for editing tables but they must be used with care as there is no way to undo the changes that they make.

There are four types of action queries: update queries, delete queries, make-table queries, and append queries. Update queries edit data in an existing table. Delete queries delete records that match the given criteria. Make-table queries create a new table consisting of records that match the criteria given. These records also remain in the old table. Append queries add records from one table to the end of another table.

To create an update query, begin a new query in design mode and then choose Update from the Query Type group on the Query Tools Design ribbon. Choose the fields you wish to edit and add the appropriate criteria and what you wish to have that field updated to. For example, suppose the name of the Design department at Gargoyle Computing was changed to Development. The following query would search out all records for employees in the Design department and change the department name to Development:

Field:	Department
Table:	Employees
Update To:	"Development"
Criteria:	"Design"
or:	

The other action queries are created in a similar manner. Choose the type of query desired from the Query Type group, and add the necessary fields and criteria.

## 4.6 Forms and Reports

### Forms

Forms are used for input into a database and to view records one at a time. Forms present data on the screen, usually one record at a time. This allows for easier input and editing of data. The easiest way to create a form is to use the Form tool. You must be in Datasheet view to do so. In the Navigation pane on the left side of the screen, highlight the table for the data you would like to use to make the form. Now, on the Create ribbon, in the Forms group, click on Form.



This creates a form containing all of the fields from the chosen table. The form is displayed in Layout View, which will allow you to make changes to the form such as resizing or repositioning the text boxes while the form is running and you can see the data being displayed on the form. Here is an image of a sample form in Layout View created from the Employees table:

Employees			
Employee Number	123-654	Date of Hire	09-Apr-08
FirstName	Nadine	Full Time	<input checked="" type="checkbox"/>
LastName	Smith	Salary	\$47,550
Department		Bonus	\$400

Record: 1 of 26 No Filter Search

To make certain types of changes and to have a wider variety of design options you will need to switch to Design View, but note that while in design view you will not be able to see the underlying data. Here is that same form displayed in Design View:

Employees			
Employee Number	Employee Number	Date of Hire	Date of Hire
FirstName	FirstName	Full Time	<input checked="" type="checkbox"/>
LastName	LastName	Salary	Salary
Department	Department	Bonus	Bonus

Record: 1 of 26

You can change the view by clicking on the View icon in the Views group on the Design or Home ribbon in the upper left corner of the screen as shown below.





You can also create a form using the Form Wizard which will allow you to choose the fields you would like to include on the form, possibly from multiple tables, and the order and style in which they are displayed. To use the form wizard, on the Create tab, in the Forms group, click on Form Wizard. The first window of the Wizard allows you to choose the table or query you wish to use as the basis for the form as well as the fields to include on the form. A form designed for input would probably include all fields. A click on the double arrow includes all fields in the form. A form for editing a database might only include those fields necessary for the user to edit the information properly. The single arrow allows the selection of fields one at a time. The second window allows the user to choose a preprogrammed layout for the form. This layout can be edited or changed later in the form creation process. The third window asks for a name. The form is now complete.

Once you have used the Wizard to create a basic form you will probably want to customize that form to better fit your needs. In the design mode you can add pictures, move elements on your form, and add common Windows elements such as labels, list boxes, or buttons. To modify the form's design either select Modify on the final window, or, if no longer in the Form Wizard, open the form in design view. In design view you will notice the window is split into three regions, form header, detail, and form footer. To add a header to your form, place your cursor on the boundary between header and detail and drag till you get a space the proper size. You can add text to your header by clicking on the Label icon in the Controls group, moving to the header area, dragging your mouse to create the desired size and then entering text. You can change the font, size and attributes of the text by clicking on the box containing the text, dragging for size, and choosing font, font size and attributes from the menus and icons above. Use the Font Color icon and the Fill/Back Color icon found in the Font group to set the text color for the label and the header background.

In the detail section of the form you can move labels and textboxes by clicking on them and dragging. You can resize any box or label. You can also change fonts and background colors using the same methods described in the paragraph above. To insert a picture into any part of your form, click on the Image icon in the Controls group, draw a box on your form, and choose a stored image from your files.

To change a field's entry box from a textbox to a listbox or menu, cut that field from your form. Now click on the proper icon in the Controls group, such as List Box. This will open a Wizard that will guide you through getting the proper entries in your box. For example, to change the Department textbox on an Entry form for Employees to a listbox, cut the current textbox from the form, click on the List Box icon and draw the box on your form. In the Wizard window, choose

to type in the values you want. Type the values Managers, Production, Sales, Purchasing, Accounting, Personnel, and Design into the boxes given on the next window. In the third window have the database store values that are selected from this list in the field Department. Finally, enter the label Department for this List Box. You will not see your list while in Design View, but if you switch to Form View, you will see the final form. You may see that some boxes still need resizing. Here's a modified input form for the table Employees:

**Employees**

## *Gargoyle Computing: Employee Data*

**Employee Number** 123-543

**FirstName** Tobias **LastName** Arnold

**Date of Hire** 09-May-98

**Full Time** ☐

**Salary** \$39,500


**Bonus** \$200

**Department**

- Accounting
- Design
- Managers
- Personnel
- Production
- Purchasing
- Sales**



Record: 1 of 42 No Filter Search

On the bottom of the form you will notice the number of the record you are viewing. You can use the arrows to move from record to record for editing, or the icon, , to get to a blank row for adding a new record.

## Reports

Reports are used to put organized data from a table or query into an attractive form for printing. Reports can be generated much as forms were; you can use the Report tool in the Reports group of the Create ribbon to generate a generic report and then customize the report in design mode.

Or you can use the Report Wizard from the Reports group to have more control over the choices of what fields you want to include and the grouping and sort order of the information being displayed. As with forms, you can choose the table or query on which to base your report. While forms are usually based on database tables, reports are usually based on a query that summarizes or chooses a limited collection of data from the database. From within the Report Wizard you can choose the table or query you want as the basis for your report and then you can select the fields you want to include for your report. The next window asks for grouping in levels for the report. A field chosen for groupings has its entries used as headings on the report. You may choose to group on more than one level. The report below is grouped by department. Underneath the grouping headings you can sort your data on any other field. The report below is sorted by date of hire. Finally, choose a layout and print style for your report and then name it. You can now edit your report in design mode. The editing process is exactly the same as for forms. You can move or resize fields, add images, and add labels with new text, if you wish.

## Gargoyle Computing:

### Salaries by Department and Date of Hire

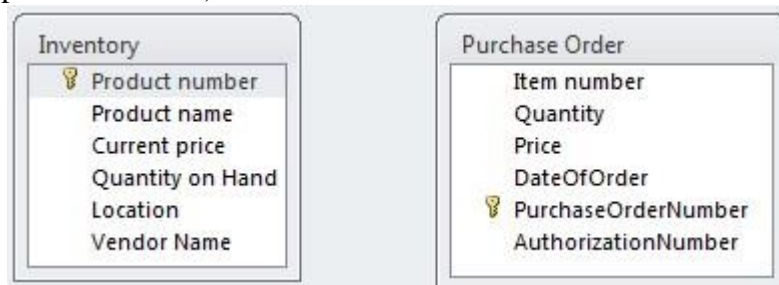
Department	Date of Hire	Employee Number	LastName	FirstName	Salary
<b>Accounting</b>					
	01-Sep-97	324-124	Madigan	Connie	\$45,700
	04-Mar-98	625-451	Jones	Diane	\$40,500
	23-May-98	123-682	Oslo	Harold	\$42,000
	19-Oct-98	893-456	Mead	Lucille	\$44,500
	05-Apr-99	459-234	Jagadinsky	Marty	\$39,700
	05-Nov-99	569-345	Conway	Robert	\$37,600
	10-Dec-99	455-256	McGonagle	Shawn	\$32,000
	30-Nov-00	789-341	Lynch	Mary	\$43,000
<b>Design</b>					
	21-Mar-97	623-678	Miller	Joan	\$61,500
	23-Apr-98	234-944	Palmer	Jane	\$55,600
	15-May-99	534-789	Thomas	Robert	\$56,250
<b>Managers</b>					
	15-Feb-94	554-234	Wangenleit	William	\$75,900

## 4.7 Relationships Between Tables and Relational Queries

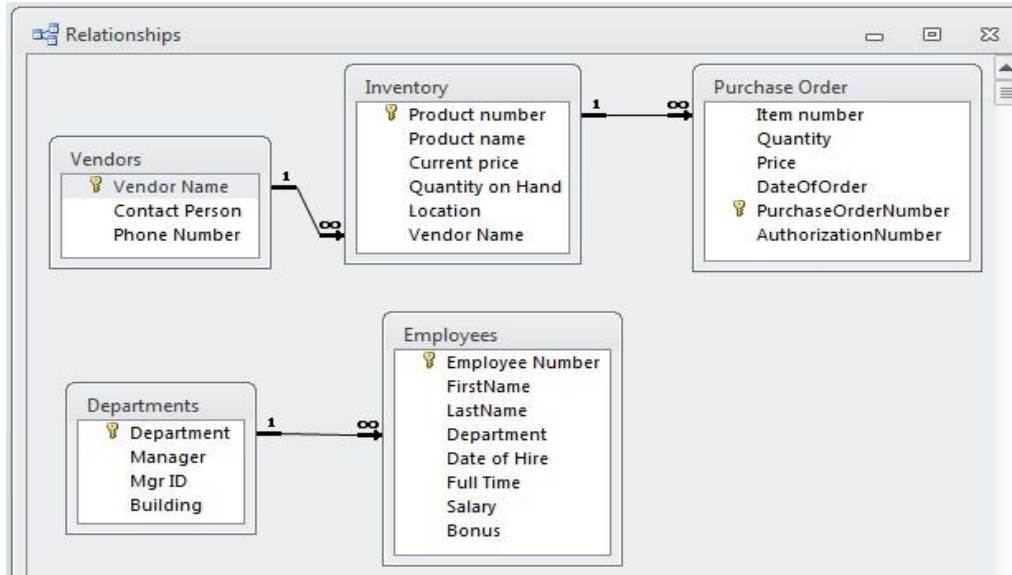
### Relationships

As we noted at the beginning of this chapter, most databases store information in several related tables, this is where the real benefit of using a relational database is realized. These tables may be linked to one another by fields that hold common data. The first step is to design your tables such that one table has a field which will map to the primary key of another table. This field is generally not part of the primary key of your table, rather it maps to the primary key of another parent table. For this reason, this column is often referred to as a “foreign key” column. Also, it does not have to have the same name as the field it relates to.

As an example of a linking field, in the two tables that follow, Product number is a key field for the Inventory table, and for each item in the Inventory table there is a unique Product number. For every record in the Purchase Order table, there is a field Item number which holds the product number for the item being ordered. Thus for any one item in the Inventory table there may be many purchase order records with that product number which suggests a natural one-to-many relationship between Product number and the Item number in the Purchase Order table. Product number would be a logical and helpful field to link the two tables. Note that the field names do not need to be the same, but the fields must contain the same data. (in this case the product number)



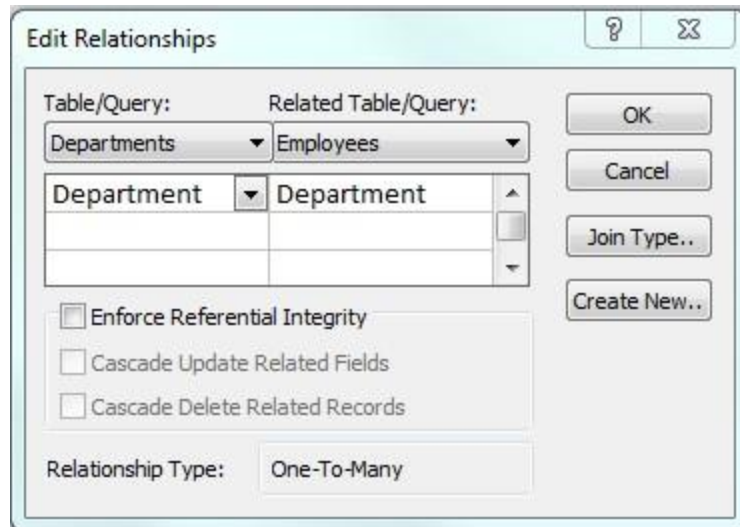
We can create relationships between multiple tables using the Relationships window in Access. Here is an example showing what that looks like after the relationships were created for the Gargoyle Computing database:



To set up the relationships between tables in the database, on the Database Tools tab, in the Relationships group, click on Relationships. Use the Show Table icon in the Relationship Tools Design ribbon to add the desired tables to the relationship window. You can also remove tables from this window with the Hide Table choice. To create a relationship between two tables, click on a field in one table and drag over the corresponding field in another table and then release the mouse. If you have a one-to-many relationship, (one parent record related to many child records) it makes the most sense to drag from the table on the “one” side of the relationship so it becomes the parent in the relationship. A window will pop up that allows you to refine the relationship. One table acts as the parent (shown on the left side of the popup window), the other as a child/related table (shown on the right side). The table you dragged from is the parent, the table you dragged to is the child.

Note: any tables involved in a relationship must be closed before you can create a relationship between them.

You will see a window like the following:



Choose “Enforce Referential Integrity” – there is rarely a need for a real database to NOT have this option selected, even though by default it is off. Selecting this adds additional notation to your relationship diagram and a “+” sign on your datasheet view on the parent table to add child records directly as shown in the following image.

Inventory						
	Product numbe	Product narr	Current pric	Quantity on Hand	Locat	Vendor Name
+	61800	P4 1800 CPU	\$765.00	1100	A-27	Johnson Computing
+	61300	P4 1300 CPU	\$687.00	700	A-27	Johnson Computing
+	49700	CD Drive	\$201.00	1200	B-42	Johnson Computing
+	49703	CD R/W Drive	\$292.50	800	B-48	IT GigaHertz

It also will prevent you from adding child rows that do not have a matching parent to prevent orphaned records. In the Gargoyle database, we would not be allowed to have an Item number in the Purchase Order table that did not have a corresponding Product number in the Inventory table. (You can’t order a product that is not in inventory.) Enforcing referential integrity insures that any record placed in the related table must have a matching record in the primary table. Checking Referential Integrity helps to maintain the accuracy (or integrity) of the information stored in the database.

Cascade Update/Delete Related Fields should also be checked typically. Cascade Update means that if the primary key of a parent record changes, the child records are automatically updated. This is useful more for parents that do not have AutoNumber primary key fields. For example, if we were to use an update query similar to the one we created in the previous section to change the name of the Design department in the Departments table to Development, cascade update would cause any reference to the Design department in the Employees table to also be changed to Development. Cascade Delete means that if you delete a parent record, Access will also delete any related child records automatically to prevent orphaned records. (E.g. if you delete Product number 123, all of the Purchase Orders with that number are also deleted).



Joins allow you to include data in queries/reports from both tables at once. You can setup the default Join type in the Relationship window by choosing Join Type. The Join Properties window will appear and then you can select the type desired, (1, 2, or 3) as seen in the following image.



A Join Type defines how queries will match up the records between the tables. This type should generally be either a 1 or 2. 1 is an “Inner Join” which means that only records that are in both the child/parent table are returned. E.g. If a Product number exists in the Gargoyle database but none of those products have been ordered, then an Inner Join query would exclude that product number from the results of a query between the Inventory and Purchase Order tables. Type 2 is an Outer Join, which means to include all records in the parent table even if they do not have any records in the child table. This would include the product that has not been ordered in the query. Type 3 is only used if you have not enabled the Enforce Referential Integrity option because it includes those records in the child table that have no match in the parent table, but if referential integrity is enabled this situation is prevented from even happening so we won’t use this option. Note, even though you define a default join type here, you can change it when you actually build multi-table queries as well.

Relationship types are almost always one-many. If you find you have a need for a many-many relationship, you usually need to add another table in between the two tables to make two onemany relationships instead.

To edit an existing relationship, just double click the line between the tables.

### Relational Queries

Relational queries (or Multi-table queries) are not much more difficult than single-table. When creating a new query, add all the tables you will require in the query. Keep in mind that all of the tables in a query **need to somehow be connected via a relationship**. When you click Close after adding the tables the relationship lines between the tables should be visible. If the relationships between the tables were not created earlier, you can create the relationship in the query design window just like you did in the Relationships window. When creating a relational query, the join type comes into play and you may need to edit the Join Type to get the desired results. If you double click the relationship lines you can change from the default Join Type defined when creating the relationship



When you choose fields note that you have fields available from both tables.

If you have the same field name in both tables, you can reference them using the table name followed by an exclamation point:

Dog![Name] & “, “ & Tag![Name]

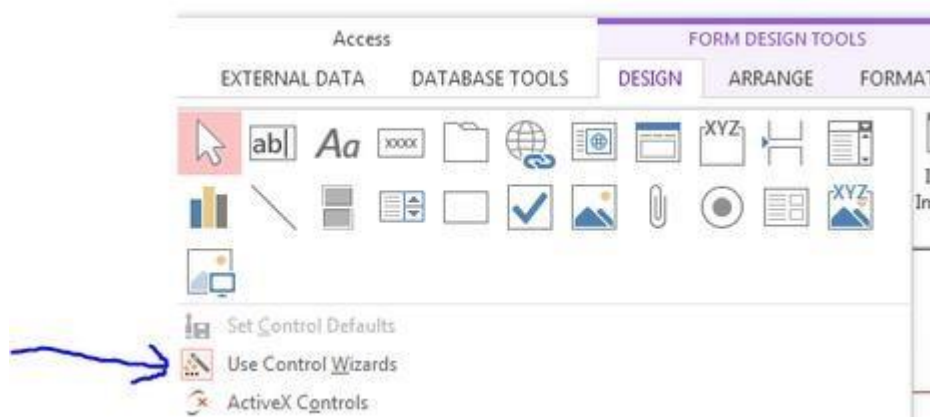
Note, it is possible to create a new query using one or more tables and an existing query. When you start creating a new query and are adding the tables desired, you can also add a query to the list of “tables” from which you will build the new query, (essentially querying on another query.)

## 4.8 *Macros in Access*

Macros can be written in a number of ways in Access. The simplest is to run a single predefined “Action” such as “Go To Previous Record” which you can assign to a button on a form/report. In addition, you can create a series of actions with the Access Macro builder to create more complicated sequences of these pre-defined actions. Finally, you can use the Visual Basic code editor to create a VBA code macro, (a VB program), where you will have full control over what actions you want to have happen in your database.

### Single Actions

The simplest way to add a “macro” to an Access form is to go to the form in Design View, then in the Controls group on the Design ribbon, choose the Button object and draw a button on your form. If the “Use Control Wizards” icon has been selected in the drop down list below the Controls group of the Design tab, when the command button is placed on the form the Command Button Wizard will pop up menus containing all of the actions available. This is the default setting but you may need to click on “Use Control Wizards” before you choose your button if that has not been checked. You may need to use the drop down menu on the right side of the Controls group in order to see the “Use Control Wizards” icon.



A Button Control Wizard will allow you to choose from one of the pre-defined Actions and guide you through some questions to create a simple button. When asked to name your button use the “btn” convention we used in Visual Basic.

A list of the Actions available with some documentation on what they do can be found at:

<http://msdn.microsoft.com/en-us/library/office/dn161227.aspx>

### **Macro Builder (list of Actions)**

If you want your form button to run multiple actions (e.g. Move to Next Record and display a Message Box) you cannot use the previous method shown, which only allows one action per button. Instead, you can use the Macro Builder to create a more complicated list of actions. From the Create ribbon, choose Macro on the right in the Macros & Code section. When clicked this brings up a macro builder which allows you to enter a list of actions. Some actions have arguments (parameters) that you can enter in the textboxes for that action. Also you can enter comments so you can remember what you were thinking! Finally, save the macro with a helpful name. Now when you add a Button to the form the Button Control Wizard again appears, but if you immediately press Cancel you can assign the button to your newly created macro. Right click on your button and select “Properties” to open the Properties window, go to the Event tab and in the OnClick event choose the name of your macro (which will be run “on click”). Two other things you should change are the Button name (using btn prefix) which is in the “Other” tab, and also the Caption, which is on the Format tab. There are numerous options to explore, but these two you should always change. Besides creating a button to run a macro created using Macro Builder, this type of macro can also be run from the menu using the Database Tools tab, Macro group, and Run Macro icon.

Note: It is also possible to simply drag and drop the Macro from the Navigation pane on the left side to the form/report to automatically create a button, saving you from doing the steps above! If you do this, it is still recommended to Right click the button, choose Properties and change the button name and caption.

To see a sample of a simple single action macro in use, open the Employees form of the Gargoyle database and click the button with the caption “Move to Next Record”. This button has a macro assigned to it, made up of one action, the “GoToRecord” action. This macro is rather limited. For example, if you are on the last record, nothing prevents you from trying to go to the next record, even though none exists. If you try to do this, you will note that an error message is generated since Access cannot perform the requested action. This is just one reason why you may want to write your own VBA macros to be able to handle situations like that in the way you want.

A common macro for databases is the AutoExec macro, which is executed (if such a macro has been created) whenever a database is loaded. Creating an AutoExec macro allows the programmer to make sure the application opens in a consistent user-friendly fashion. The programmer includes as many actions as desired to get the application ready for the end-user.

Typically, a form is displayed that introduces the user to the application and lists the options available to the user.

## **VBA Macros**

Built-in macros are useful for many common tasks, such as navigating from record to record, adding or deleting a record, or sorting records into a desired order. However, often a user would like to do tasks for which there are no built-in macros available. The user can design and write his/her own VBA code within the Access database to accomplish a desired task and then assign that code to be executed on the “On Click” event of a button on a form or one of many other events. If you would like information about writing VBA macros within Access, see the section at the end of Appendix C. Rather than writing VBA macros within Access, in this course we are going to focus on connecting to an Access database from within a Visual Basic program, building on the VB skills you have already developed. See the next section.

## **4.9 Connecting to an Access Database from within a VB program**

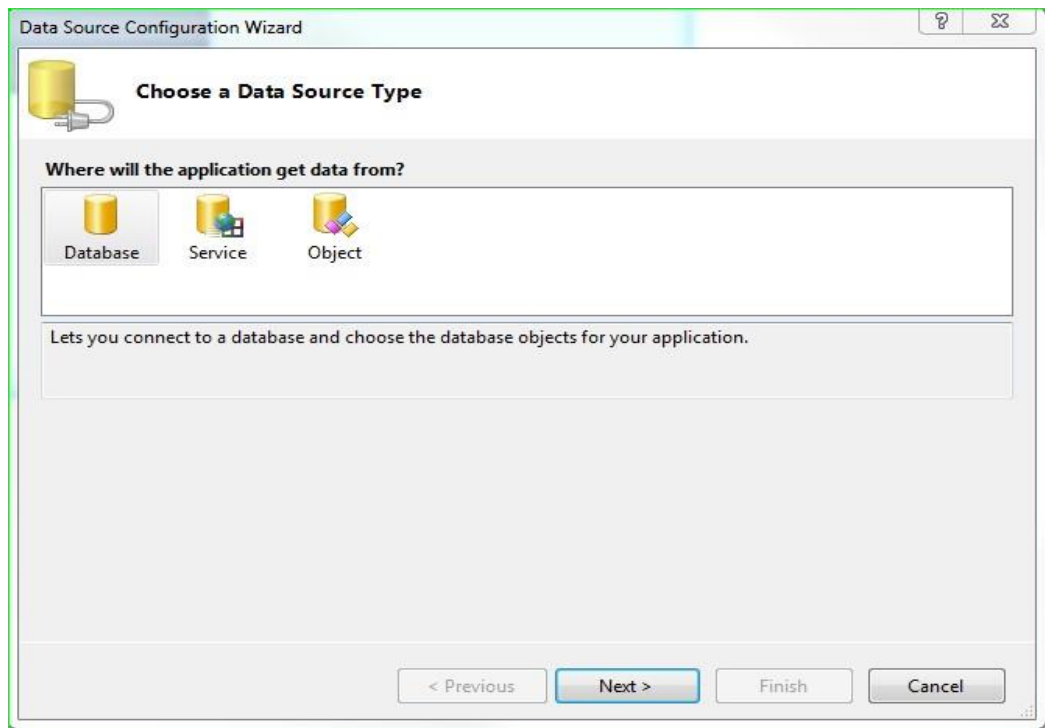
It may be more desirable to write a VB program that interacts with a database rather than writing a VBA macro as part of the database. This section will show you how you can display and/or search for information that is in either a table or in the results of a query from an Access database.

### **Creating a Database Connection:**

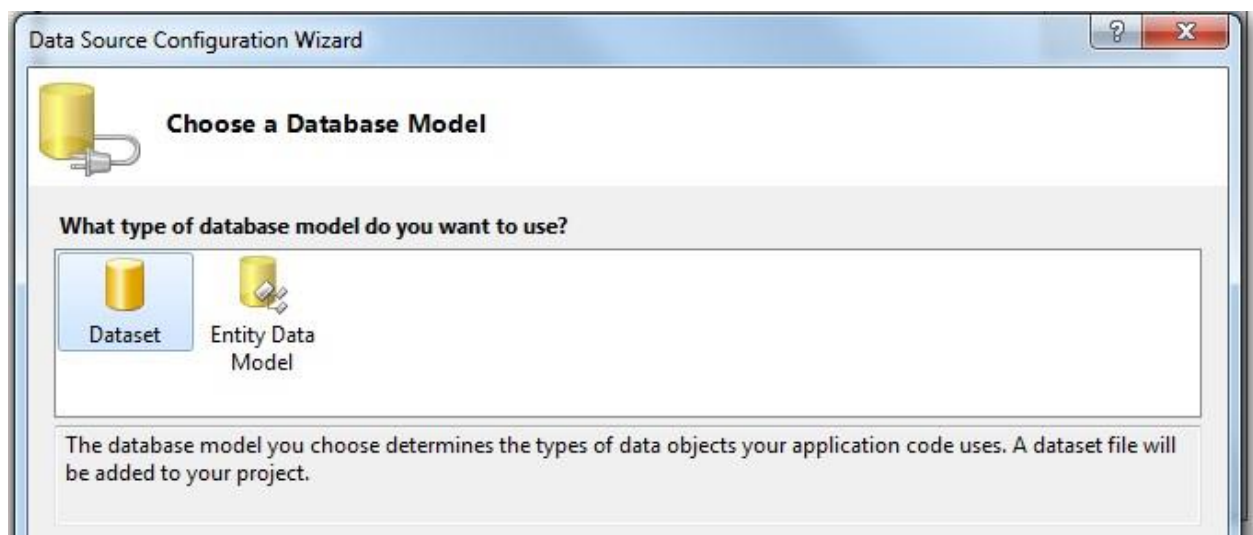
This section walks you through the steps to create a VB program that connects to an Access database and shows how you can get information from the database.

Start a new VB project with a blank form and save it somewhere on the M: drive within your CS130 folder, calling it something like *databaseConnectionProject*.

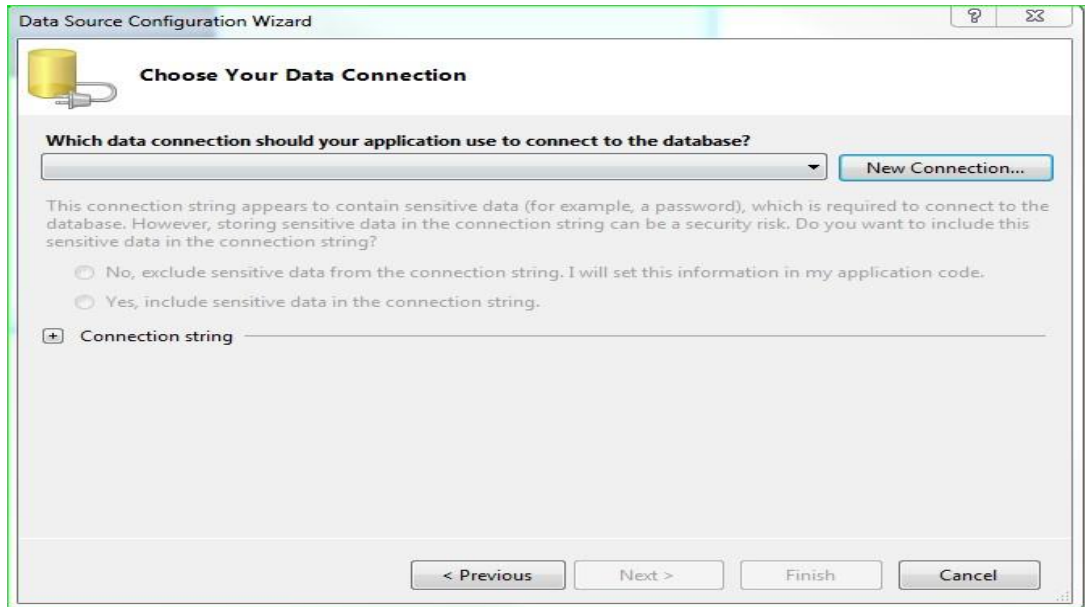
In order to work with an Access database from within VB, we first need to setup a connection to a database file. Look at the left side of your VB window next to the *Toolbox*; if there isn't a tab named *Data Sources* then add it by selecting *Data > Show Data Sources*. Once you have the *Data Sources* tab visible, press on it to expand it (if not already expanded), then right-click anywhere inside the *Data Sources* area and select *Add New Data Source* which should display the following window.



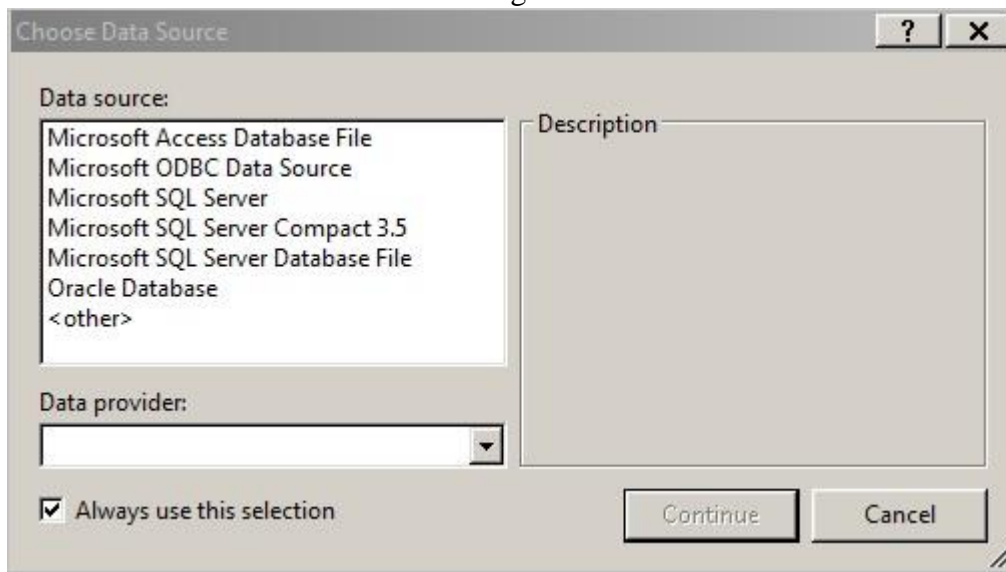
Select *Database* and press *Next >*. That will bring up the following window:



Select *Dataset* and press *Next >* which should display the following window:



Press on the *New Connection* button to get the window shown next:



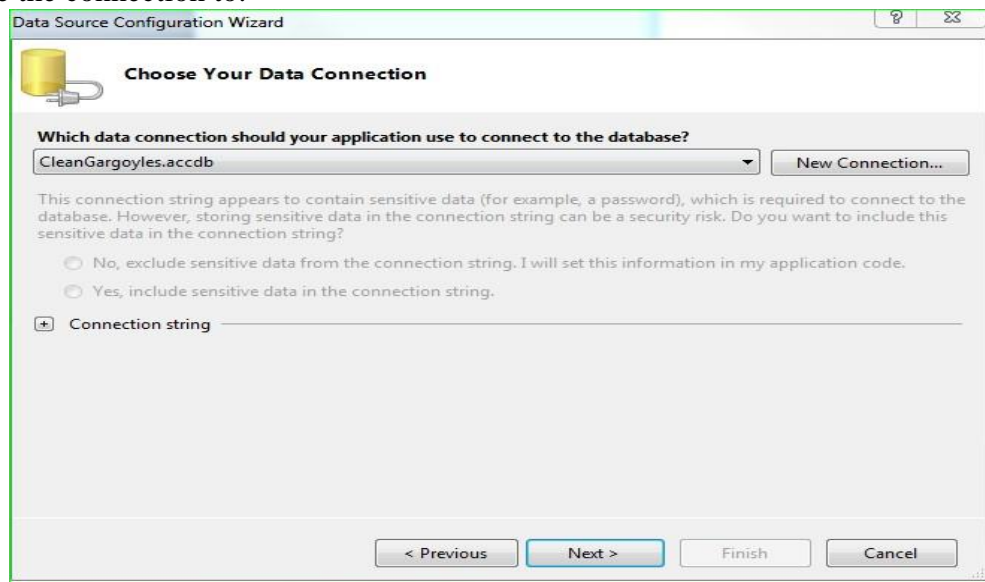
Select *Microsoft Access Database File* under *Data source* and press *Continue*.

This will take you to the window shown next:

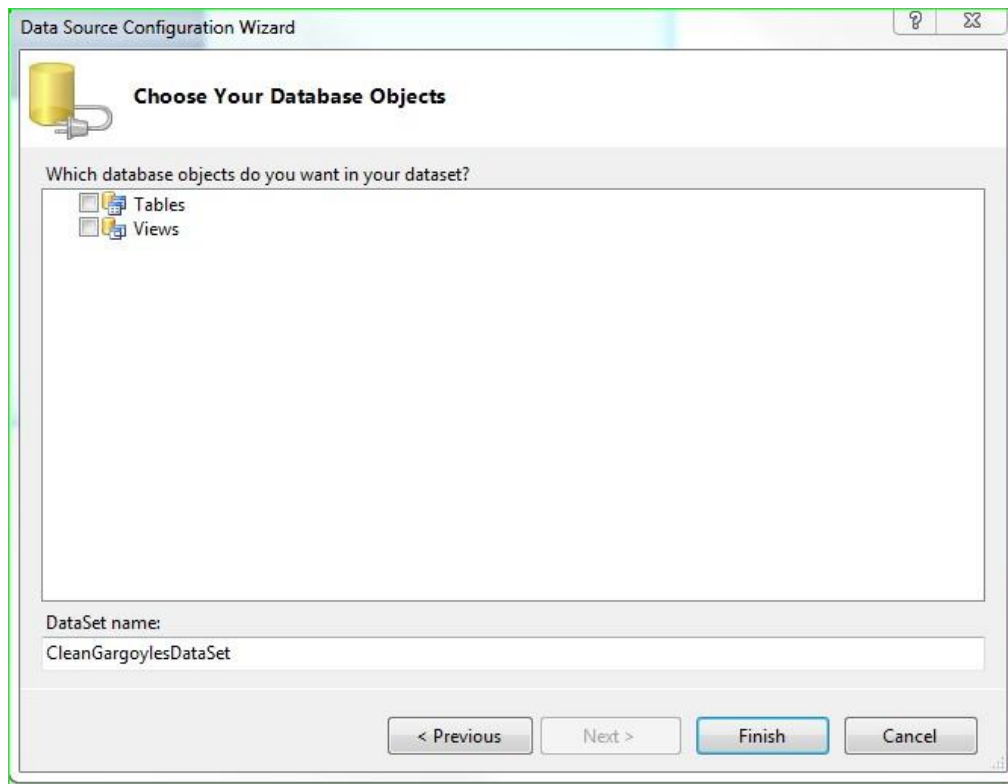


Now press on the *Browse* button under *Database file name* and navigate to the database file you would like to connect to, (in this sample *CleanGargoyles.accdb* was chosen). When done, press *Test Connection* and then *OK*.

This brings up the window below that now displays the name of the Access database file you have made the connection to:



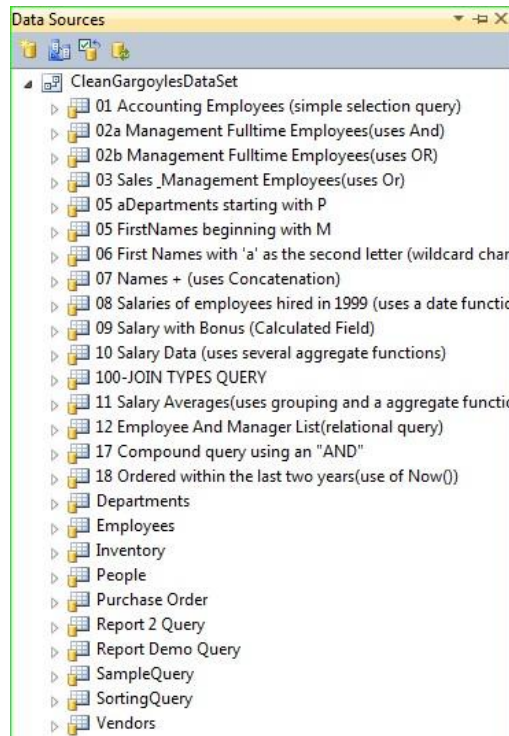
When you press *Next >* on the window above, VB will display a popup box asking you to save a local copy of the database; press *Yes*. The window that now pops up asks if you would like to save the connection string to the application configuration file. You can just accept the default name and press *Next >* to continue. This will pop up yet one more window where you choose the database objects you would like to be able to use:



Now check the *Tables* and *Views* options and press the *Finish* button.

At this point you have made a connection to a database and you should see in your *Data Sources* area, a list which includes all tables and queries in the selected database.



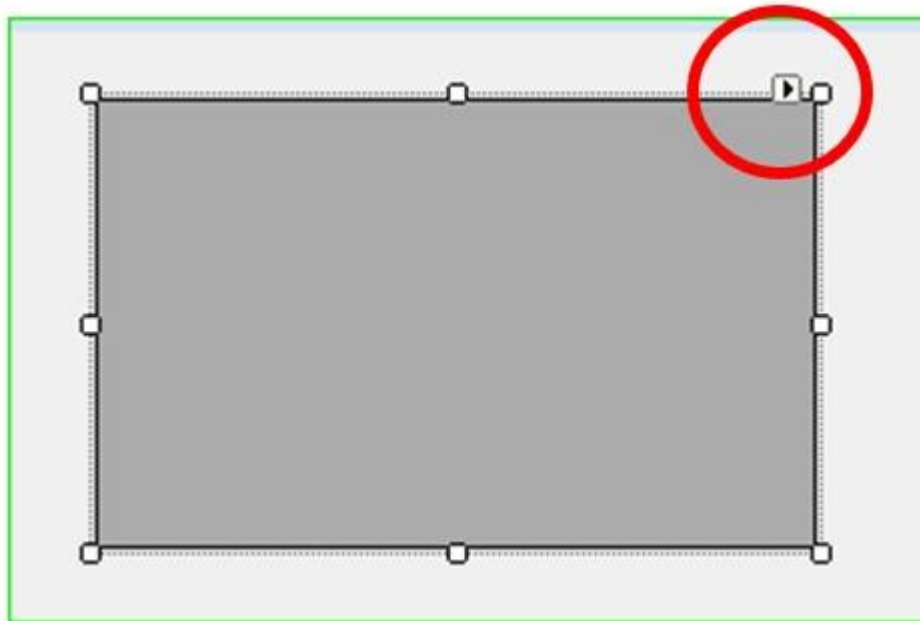


You can now view all the data stored inside the selected database from within VB without having to launch Access; for example, right click on table *Departments*, select *Preview Data* and then press on the *Preview* button in the window that appears.

Establishing the connection to the database is the first big step in creating your VB program to connect to a database. The second big step is introducing a way to be able to refer to the individual data items by name. To do that we will be adding a new object from the VB toolbox, a *DataGridView*, to the form in our VB program and choosing one of those data sources to be associated with it. As the name suggests, a *DataGridView* is an object that makes it convenient to look at data in a grid format with rows and columns.

### **Setting up a DataGridView Object:**

Go back to your VB form design and add a new *DataGridView* object from the *Toolbox*. In the *DataGridView Tasks* window that appears (if none appears, press on the little black triangle in the upper right corner of the *DataGridView* object). We will use this window to bind the *DataGridView* object to one of our tables or queries.



Press on the drop down menu next to *Choose Data Source*, then expand *Other Data Sources*, *Project Data Sources*, and yourDatabaseNameDataSet, (for this sample that was *CleanGargoylesDataSet*), respectively, and finally select the table *or query* you would like to use, (for this sample, the *Departments* table was chosen.) When done, make sure *Enable Column Reordering* is checked and then press anywhere to close the *DataGridView Tasks* window. Resize the *DataGridView* object to properly display all columns.

Run your VB project to see how the *DataGridView* object displays the data. It should like something like below:

	Department	Manager	Mgr ID	Building
▶	Accounting	Rahali	4567321	A
	IT Services	John	11111	C
	ITS	Mike	111	V
	Janitorial	Smith	982344	B
	Management	Wangenleit	554234	A
	Personnel	Winkler	456324	B

You can sort the data by any of the columns by simply pressing on the column name (press the column name twice to sort in descending order).

Stop your VB project and go back to the form design view. Rename your *DataGridView* object using the naming convention for *DataGridView* objects of starting their names with the letters *dgv*, (ie *dgvDepartment*), and change the *AllowUserToAddRows* property to False.

Once you have added a *DataGridView* object to your form and bound it to the desired data, it is finally time to add a command button and a *RichTextBox* and to write some VB code to do something with the data you have connected to. Add an appropriately named button object, (ie. *btnFindDepartmentsByBuilding*), and a *RichTextBox* object called *outResults* to your form before you begin to write the code for your project.

### **Writing code to use the DataGridView:**

You can think of a *DataGridView* as a two-dimensional container that holds the set of data that you associated with it, essentially a table having rows and columns of data. If we simply wanted to see all of the data in a table, we would not have to write any VB code, because once a *DataGridView* has been bound to a *datasource*, when you run your program, the data is displayed in the grid. It is particularly useful to use a *DataGridView* when you would like to search for data or display only a portion of the data. The *DataGridView* has a property called *RowCount* that holds the number of rows. We will use the *RowCount* property to set up a For/Next Loop to iterate over all of the rows in the table and will use the *Rows* and *Cells* methods of a *DataGridView* to refer to a particular item of data by indicating the row number and column number the data is in. The row and column indices start at 0, (ie. *dgvDepartment.Rows(2).Cells(1).Value* would refer to the data in the third row and second column of the table.) Below is the code that would display the names of all of the departments in the *Departments* table, assuming *dgvDepartment* is a *DataGridView* bound to the *datasource* for such a table and the first column in that table is the name of the department.

```
Dim position As Integer = 0

For position = 0 To dgvDepartment.RowCount - 1
    outResults.AppendText(dgvDepartment.Rows(position).Cells(0).Value & vbNewLine)
Next position
```

The code above displays all department names in the *RichTextBox* object.

Note the following:

1. Looping over a *DataGridView* object: The For loop structure used to loop over the rows in a *DataGridView* object requires a variable (i.e. *position* in our case) to start at 0 and go up to one less than the number of rows in the *DataGridView* object (i.e. *dgvDepartment.RowCount-1* in our case)

2. Accessing data in a *DataGridView* object: To access a specific column value of a given row we need to use row and column indexes as follows:

**NameOfDataGridViewObject.Rows(index of desired row).Cells(index of desired column).Value**

For example to access the *Building* column (the 4<sup>th</sup> column which has an index of 3 because indexes start at 0) of the second department (which has an index of 1 because indexes start at 0) we use `dgvDepartment.Rows(1).Cells(3).Value`

Following this paragraph is the code for a sample VB program that connects to an Access database that contains information about a computing company, “Gargoyle Computing”. The program contains a *DataGridView* that is bound to the “Department” table. The program asks the user for a department name and then searches the database for the name of that department and if it is found, it displays the name of the manager of that department, essentially querying the database and using the field names to search for and retrieve information. If that department name is not found, an error message is displayed.

```
Private Sub btnFindManager_Click(sender As System.Object, e As System.EventArgs) Handles
    Dim deptName As String = "", found As Boolean = False, position As Integer = 0
    outResults.Clear()

    deptName = InputBox("Please enter the name of a department.", "Get Department Name")

    For position = 0 To dgvDepts.RowCount - 1

        If deptName = dgvDepts.Rows(position).Cells(0).Value Then
            found = True
            outResults.AppendText("The manager of the " & deptName & " department is " &
                                dgvDepts.Rows(position).Cells(1).Value)
        End If

    Next position

    If Not found Then
        outResults.AppendText("Sorry, that department name is not in the table.")
    End If
End Sub
```

---

#### ***4.10 Conclusion***

Databases are an important tool for storing and managing large sets of related data. Databases centralize a user's information and provide a variety of methods for data entry, editing, and retrieval. The real power of databases comes from the use of queries to search through the data and select carefully chosen subsets of the database to be displayed or printed as part of a report. The use of referential integrity and cascade update and delete insures that a database will be maintained accurately and that tables will include only relevant records. These options provide a measure of control and security, ensuring that the information in the database stays up to date and consistent across all tables.

Database files can be accessed by computer programs other than the database management software used to create the database. Visual Basic, as well as many other computer programming languages, have components of the language designed to make connecting to a database much easier.

# Chapter 5

## Spreadsheets: An Introduction to Excel 2016

### 5.1 Introduction to and Uses of Spreadsheets

Spreadsheets are among the most useful applications packages, both for individual use and in a business setting. Spreadsheets are used to organize, store, calculate with, and display primarily numeric data in tabular and graphic formats.

A spreadsheet is basically just a large piece of graph paper, divided into rows and columns. Each cell can hold either a piece of data or a formula. The power of a spreadsheet comes from its ability to use data in formulas and recalculate these formulas as the data changes. When a piece of data used in a formula is changed, the formula is automatically recalculated. For example, suppose a professor were storing students' grades in a gradebook. The professor has calculated the average score on each exam and also each student's average up to this point in the term. Suppose a student finds an error in the grading of his exam and asks the professor to change the grade in her gradebook. If the gradebook were kept manually, the professor would have to change the student's grade entry and then recalculate the average on that exam and that student's average so far. On a computerized spreadsheet, the professor would only have to change the grade entry. The exam average and the student's average would be automatically recalculated by the spreadsheet.

The figure below shows such a spreadsheet. Notice that the rows are numbered while the columns are designated with letters. Some cells, such as those in rows one and three, hold character data. The cells in columns C through H hold numeric data. The cells in columns J and K hold formulas. Cell J4 is highlighted. On the worksheet in J4 you see the result of the formula, while the formula is shown directly above the worksheet.

	A	B	C	D	E	F	G	H	I	J	K	L
1	Final Grades: CSCI 190											
2												
3			Exam 1	Exam 2	Exam 3	Final	Quiz Total	Labs/Homework		Overall Percentage	Grade	
4	Ambrose	Sandy	89	71	85	90	92	92		88.50	B	
5	Buck	Linda	94	96	88	93	89	88		90.60	A	
6	Case	Carmen	85	83	87	88	77	75		81.00	B	

Spreadsheets are used for any application that calls for the manipulation of numeric data. They are appropriate for most accounting tasks. Spreadsheets are also useful for business analysis and

forecasting. Because one can change entries and have those changes reflected automatically in the formulas on the sheet, spreadsheets are good for ‘what if’ analysis. A user can enter data for a variety of scenarios and see how those data change their bottom line profits or losses. Spreadsheets are also useful for any applications in which data is to be presented in graphical format.

## 5.2 *Entering and Formatting Data*

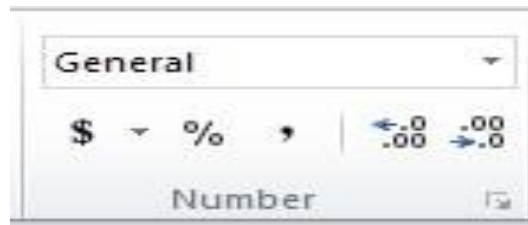
Each project in Excel is called a workbook. To begin a new workbook, just choose the **Blank workbook**, (or another of several workbook templates displayed when you launch the Excel program.) or if you are in an existing workbook already you can use the menu, and choose File -- New -- Blank Workbook. A blank workbook contains a single blank worksheet in it. Additional worksheets can be added as needed by clicking on the “+” sign at the bottom of the screen. Various types of data can be entered into the cells of these sheets as you develop your project. A single cell might hold any of several types of entries: text, numbers, formulas, or functions.

### *Text*

Text is entered simply by highlighting the desired cell and typing in the text. You can enter a string of characters of up to 32,000 characters and this string will be stored in the selected cell. The entire string will be displayed in the box above the worksheet. However, the entire string may or may not be displayed on the worksheet itself. If the text is longer than the width of the cell, it will extend into the neighboring cell, but only if that cell is empty. By default, all text is left-aligned, though you can change this by highlighting the cells you wish to change and clicking on the appropriate alignment button. You can add boldface, italics or underlining in the same way.

### *Numbers*

Numbers are entered in the same way, by clicking on the appropriate cell and typing in the number. Numbers are right aligned by default. They can be formatted for currency, percent, commas, and number of decimal places using the format buttons found in the Number group of the Home menu ribbon.



Some other quick options for formatting can be found by highlighting the cell and then clicking on the top drop-down menu (General) within the Number group on the Home ribbon or if you want to see all of the possible options for formatting, click on the arrow on the lower right side of the Number tab, (the dialog box launcher used in many of the tabs). Another convenient option is to highlight a cell (or range of cells) and right click, then choose Format Cells and make your choice from a list of all of the formatting options.



If a number is too long to be displayed in a cell, the spreadsheet will either convert it into scientific notation or display a row of pound signs. Columns or Rows can be re-sized using your mouse – place your mouse on the line between two rows/columns headers and drag when your cursor changes to a line with two arrows. You can also click a row/column header and right click to choose either Row Height or Column Width and then enter the desired size. A very convenient way to autosize a column to fit the contents of the longest entry in the column is to double click the right side of column header.

### ***5.3 Cell References, Functions, and Formulas***

#### *Individual Cell References*

Spreadsheets are designed to calculate or manipulate the data contained in the cells of a worksheet. Cells are referred to using the combination of the column and row the cell is in. E4 refers to the cell in column E, row 4. The reference E4 is called the cell address or cell reference.

#### *Range References*


Often in using a spreadsheet a calculation is required that involves a group of cells, say for example finding the sum of a column of numbers. The cells could be referred to individually by their address, but if we are adding very many cells together the expression would become very lengthy and tedious to type in, (e.g.,  $B5 + B6 + B7 + B8 + B9 + B10 + \dots$ ) To refer to a group of cells in a more convenient way, a Range reference is used. A Range of cells is any rectangular group of cells, including a single cell, and the entire group can be referred to by using the cell addresses of the diagonally opposite corners of the group, separated by a colon (e.g., B5:B10)

Cell references are used in creating functions and formulas to calculate/manipulate data in the spreadsheet as you will see in the next section. The beauty and genius of using cell references in a spreadsheet is that when the contents of a cell is changed, the function or formula will automatically recalculate using the new value found in that location.

#### *Simple Functions*

A common task on a spreadsheet is to find the total of a row or a column. While you could write a formula such as  $=C1 + C2 + C3 + C4$ , there's an easier way. As soon as you enter the = sign, notice that the name box in the upper left corner changes to a listing of available functions. If the function you want is not on the visible list, clicking on More Functions gives a listing of all the built-in functions available, with a brief description of each function, as you highlight it. To find a total for a column, click on Sum. The spreadsheet will guess at the range you wish to add. If it is incorrect, change the range values suggested, then click on ok. Your formula will begin with the equal sign and then the function name, followed by the range, e.g.  $=SUM(C1:C4)$ . The range for a function might be cells in a row, a column, or even a block of cells such as C1:F5. Other common simple functions frequently applied to a range of cells include AVERAGE, MAX, MIN, and COUNT. Excel has many built-in functions. You can explore them by category from the Formulas menu ribbon, in the Function Library group.

## Formulas

Formulas must begin with an = sign. To begin entering a formula you can either type = or click on the  above the worksheet and choose a function. Formulas may contain constants, cell references, cell ranges, and functions. These are connected with the standard arithmetical and logical operators: +, -, \*, /, ^, AND, OR, NOT.

An example of a formula using a function, constants and cell references is on the spreadsheet pictured on page 5[1]. The contents of cell J4 is the formula:

$$=SUM(C4:E4)*0.1+(F4+G4)*0.2+H4*0.3$$

This formula calculates the final grade by weighting the exams as 10% each, the final and quiz total as 20% each, and the labs/homework as 30%. If the contents of any of the referenced cells were changed this formula would automatically be recalculated and the change reflected in the number displayed on the worksheet in cell J4.

Once a formula is written, we often wish to use the same formula for other rows or columns on our worksheet. In this example, we would like a similar formula,  $=SUM(C5:E5)*0.1+(F5+G5)*0.2+H5*0.3$ , in cell J5, to compute the final grade for the second student. We need not write out a new formula. We can copy the formula from J4 into J5 in two ways. The simplest is simply to place the cursor on the small square in the lower right corner of J4. A small black cross will appear in place of the mouse cursor. Holding down the left mouse button, drag that cross down the column to J8. This will copy the formula from J4 to cells J5 through J8. For each row the formula will be automatically altered to use the cells from columns C through H from that row, i.e. F4 becomes F5 for the formula in J5, F6 for the formula in J6, etc. You can also copy formulas by highlighting the cell the formula is in and then choosing Copy from the clipboard tab on the menu and Paste it in the cells desired.

An important thing to note about formulas is that you never want to use constants in them if those constants are already in a cell in the worksheet. Use cell references whenever possible. There are two reasons for this. First, since many formulas are copied down a column or across a row, you want the formula to be as general as possible. If we had used the first student's scores instead of cell references (e.g.,  $=SUM(C4:E4)*0.1+(90+92)*0.2+92*0.3$ , we could not have copied the formula into other rows without making changes to the formula. Also, one of the main strengths of a spreadsheet is its ability to recalculate all formulas when a piece of data is changed. If we changed a score for a student, a formula using a reference to that score would reflect the change, while a formula using all constants would not reflect that change.

## Relative and Absolute Addressing

### Relative Addressing

In the example above, when we copied the formula down a column from cell J4 into cells J5 through J8, all cell references were altered relative to where the formula was copied to. Copying from J4 to J5 meant that F4 changed to F5, G4 changed to G5, etc. The Overall Percentage formula was doing “relatively the same calculation” for each student. The cells being referred to were in “relatively the same” location in relation to where the formula is for a particular student. In Excel when functions or formulas are copied, the cell references are adjusted automatically to refer to “relatively the same” locations in relation to the position of the original formula.

Remember this software was designed initially for accounting purposes where the most common activity was to do a calculation with a row or column of numbers and then repeat that on many rows or columns. Having the cell addresses adjust when copying is done makes it very simple to write a single formula once and then copy it to all of those locations where you are doing “relatively the same” calculation. This is called *relative addressing*. The cell references change relative to where the formula is located on the worksheet. Copying a formula across a row produces the same effect. Suppose the formula =A5 + A6 was in cell A7. Copying this formula to B7 would change it to =B5 + B6. Copying this formula to C9 would change it to =C7 + C8; we moved over two columns and down two rows so every cell reference also moved over two columns, from A to C, and down two rows, from 5 and 6 to 7 and 8. Relative addressing often does exactly what we want it to do for us, but sometimes we do not want this change to occur. Consider the following worksheet:

	A	B	C	D	E	F	G
1	Kinderstop Day Care Center						
2	St. Joseph, Minnesota						
3							
4							
5	Weekly Rates per Child						
6		I. Age three months to kindergarten			\$ 475.00		
7		II. Kindergarten age			\$ 350.00		
8		III. After school care			\$ 140.00		
9							
10							
	Parent		Number of				Total
11	Name		Children	I	II	III	charge
12	Johnson, Nancey		3	1			2
13	Smith, George		1		1		
14	Thompson, Richard		1	1			
15	Llewellyn, Sandra		2		1	1	
16	Thorson, Kirsten		2		1	1	

### *Absolute Addressing*

The formula to compute the total charge for cell G12 might be =D12\*E6 + E12\* E7 + F12\* E8. This formula multiplies the number of children in each category by the charge for that age group. Notice the use of cell references here, for example, E6 rather than 475. This would allow the company to change the charge for young children without having to change any formulas.

If we were to copy this formula down the column, however, we would encounter a problem. The formula would copy into G13 as =D13\*E7 + E13\*E8 + F13\*E9. All the cell references move down one. But we don't want the references to E6 through E8 to move at all. To hold these references in place, we must add \$ signs to the reference, making these *absolute references*. A \$ before the row number makes the row absolute, meaning no matter where you copy that formula to, the row will not change but the column will. A \$ before the column letter makes the column absolute, and a \$ before both means the cell reference will never change, no matter where the formula is copied to. So what we really want for our formula in cell G12 is:

=D12\*\$E\$6+E12\*\$E\$7+F12\*\$E\$8

The use of the \$ in front of both the column and row in the cell reference makes it an Absolute Reference that will not change at all when copied.

### Range Names

A cell that contains a value frequently used in a formula can be assigned a name. By naming a cell, you can refer to that cell without having to remember the column letter and row number. For example, in the above formula we might have:

=D12\*ToddlerRate+E12\*KinderRate+F12\*AfterSchoolRate

Here are two of the many ways to name a cell or a group of cells. Using the menus, select the cell and then choose the Formulas ribbon from the menu and then use Define Name from the Defined Names group. Type in your name and select OK. You will notice that whenever you now select this cell, the associated name will appear in the name box at the far left, directly above the worksheet. You can also give a single cell or a group of cells a name easily by highlighting the cell or cells and then clicking in the name box, (box in upper left of worksheet that displays the address of the cell), typing in the desired name, and then pressing the “Enter” key to create the range name. An important thing to note about Range names is that they are Absolute References and can be referenced from any worksheet of the workbook simply using the range name.

### Multiple Sheets

Data can be stored on separate pages or worksheets within the same workbook. Each new workbook that is created begins with a single sheet, designated Sheet1. You can add more sheets by clicking on the “+” sign at the bottom of the screen, just to the right of the name of the last sheet.

Clicking on the tab for a sheet brings that sheet to the foreground. You can delete sheets by



clicking on that sheet's tab and then use a right mouse click to bring up a menu to choose Delete from. Deleting a sheet that is referenced by formulas in other sheets will cause errors in those formulas.

When a cell is referenced in a formula by only a row number and column letter (e.g. E8) it refers to that cell on the current worksheet. To reference a cell on a different sheet, you add the sheet name followed by an exclamation point (e.g. Sheet2!E8 or =Sum(QuizGrades!E8:E12)).

You can rename the worksheets with appropriate names for your application by double clicking the tab holding the name of the sheet and typing in the new name. If you rename a sheet it updates any references in formulas to the new sheet name. You can rearrange your sheets by clicking on a tab, holding down the button, and dragging that tab to the location you desire.

### Printing a Worksheet

When preparing to print you may want to make some choices from the Page Layout ribbon to change some of the settings such as margins and paper orientation. Worksheets often fit best in Landscape mode. You can choose whether to include the gridlines and row and column headings using the Sheet Options group on the Page Layout ribbon. To print only a portion of your worksheet, you can either highlight the cells you wish to print each time you print or you can set

a “print area” that will be remembered when you save your worksheet. To set or clear the print area, choose Print Area from the Page Setup group on the Page Layout ribbon. If no cells are highlighted and a print area is not defined, it is assumed you wish to print the entire worksheet. Once you have decided what to print, you may wish to use the menu system and choose View->Page Break Preview to see how it will look. The page breaks can be easily moved around to the desired location while in Page Break Preview. When you are ready to print, use the system menu and choose File-> Print and then make the appropriate choice under “Settings”, whether printing the Active-sheet, selection, or entire workbook. If this preview looks good, click on **Print**.

## 5.4 *Sorting and Conditional Functions*

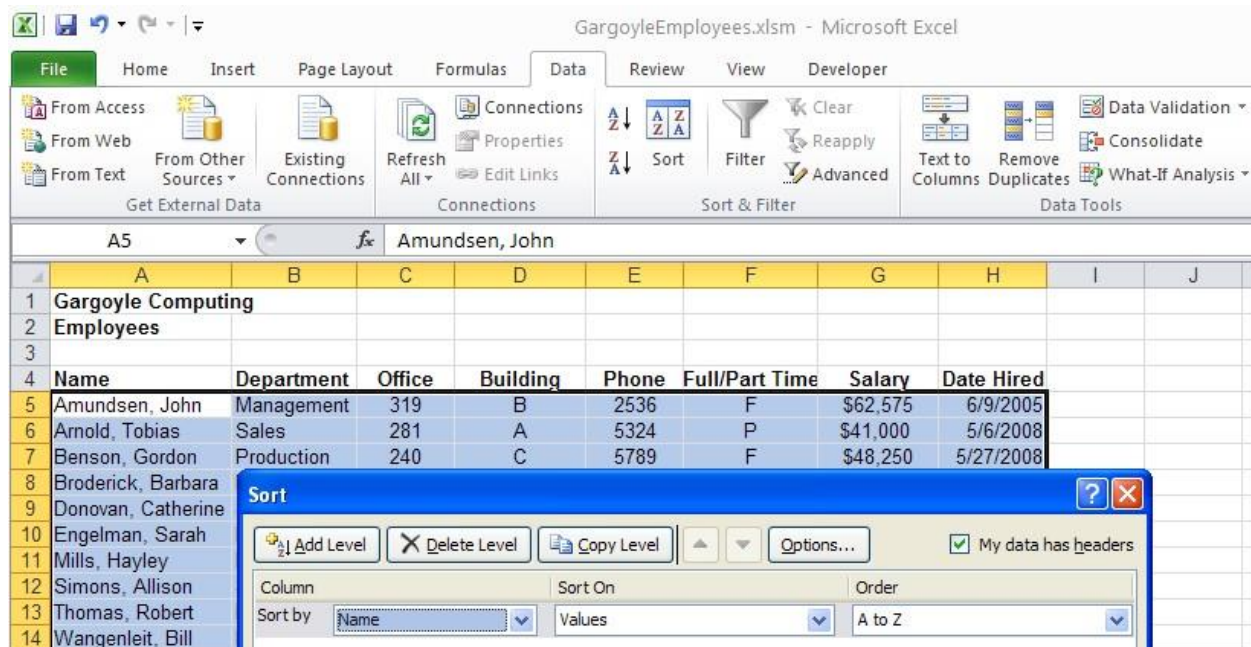
### Sorting

Sorting is quite easy in Excel. Consider the following worksheet:

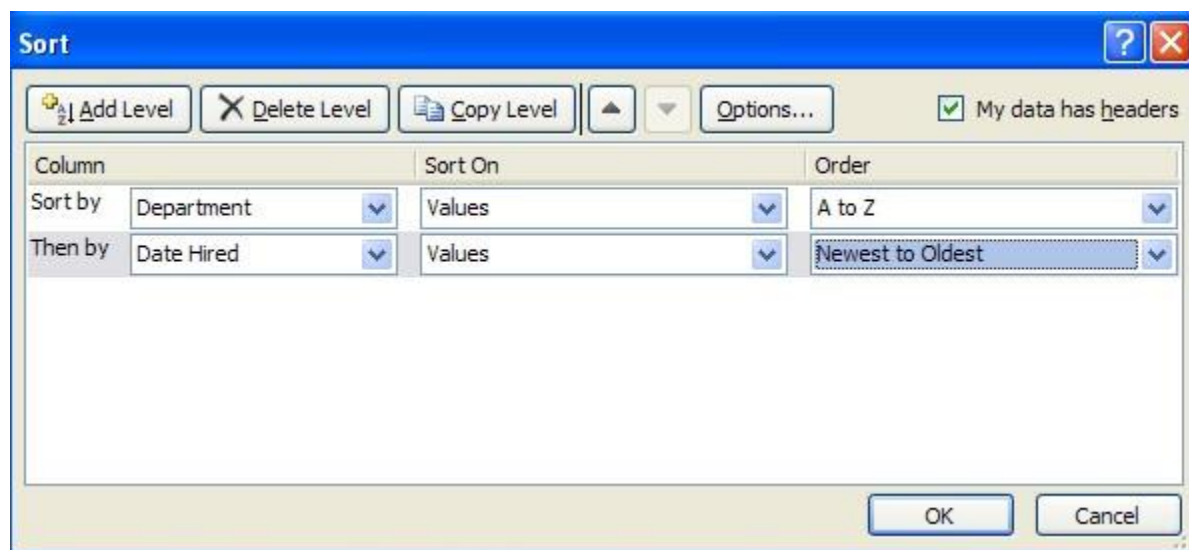
	A	B	C	D	E	F	G	H
1	Gargoyle Computing							
2	Employees							
3								
4	Name	Department	Office	Building	Phone	Full/Part Time	Salary	Date Hired
5	Mills, Hayley	Design	240	B	3287	F	\$65,500	3/19/2007
6	Thomas, Robert	Design	210	B	4278	F	\$57,250	5/12/2009
7	Wangenleit, Bill	Management	318	B	2689	F	\$75,900	2/13/2004
8	Amundsen, John	Management	319	B	2536	F	\$62,575	6/9/2005
9	Winston, Avery	Management	215	B	4374	F	\$63,500	6/12/2005
10	Broderick, Barbara	Production	249	C	5214	P	\$21,500	1/13/2008
11	Benson, Gordon	Production	240	C	5789	F	\$48,250	5/27/2008

The information on this sheet could be sorted on a variety of keys, such as name, department, salary, or date hired. To sort on a single column, for example, by name, simply click on a cell in that column and then click on the AZ icon in the Sort & Filter group on the Data ribbon. The information in all columns will be moved, sorted alphabetical by name. If your data has a header row, like the data in the worksheet above, you may need to first choose the Sort icon and indicate, by checking a box in the Sort window and indicating the column and order that you wish to sort on before proceeding with the sort. One of the problems that can occur when sorting happens when the data is not contiguous (a blank column somewhere in the middle.) The software makes the assumption that the data to be sorted is always contiguous and sorts only that data. When your data is not contiguous, you should highlight all of the data you want to sort, (including the blank column(s)) and then from the Data ribbon, in the Sort and Filter group, click the Sort icon and make your choices of how you want the data sorted in the pane that appears.





Sometimes we wish to sort on two keys. To sort all employees by Department, and within each department by date hired using the Sort icon in the Sort & Filter group, sort first on Department, (your primary key), and then add another level and sort on Date Hired, (your secondary key).



You can also get to sort options in a couple other ways. Simply right-clicking will bring up a Sort option that will take you to whatever options you want or from the Home ribbon, within the Editing group, you can click on the AZ icon which will bring up sort options. If you only want to sort a portion of your data, highlight only the data you wish to sort. If you don't highlight your data, the program will make the assumption that all contiguous columns are to be sorted.

## Conditional Functions

Excel has several built-in conditional functions that allow different actions to be made, depending on the data. Two of the more popular ones are the IF function and VLOOKUP function. The IF function has the same purpose as an If-Then-Else statement; choose one of two actions, depending on a condition. The VLOOKUP function allows the user to select one of multiple options that are listed in a separate table in the spreadsheet, depending on the value of the parameters passed to the function. In effect, it “looks up” the desired result in a table that has been previously created within the spreadsheet, (e.g., One could use a VLOOKUP function to find the zip-code of city if a table containing the names and zip codes of the desired cities is available in the spreadsheet.)

### *IF Function*

The If function enters one of two possible results in a cell, based on whether a given condition is true or false. The format for an If function:

`=If(<condition>, <action if condition is true>, <action if condition is false>)`

On the spreadsheet on the previous page, suppose workers who were full time were eligible for a bonus equal to ten percent of their salary and we wished to enter the bonus amount or “no” in column I depending on this eligibility. In cell I5 we would enter the function:

`=IF(F5 = “F”, G5*0.1, “no”)`

The first parameter is the condition. This condition must be an equation or an inequality, (a Boolean expression that can be determined to be true or false). The second parameter is the result that will be entered in the cell if the condition is true and the final parameter is the result if the condition is false. The *If* function in Excel serves the same purpose as the If-Then-Else statement in a programming language. An If-statement in Visual Basic that is equivalent to the IF function above might look like this:

```
If F5 = “F” then
    Print G5 * 0.1
Else
    Print “no”
End If
```

We can also use AND or OR to form compound conditions. For example, if we would like to give only full time Production Department employees the ten percent bonus, then the condition would be that the worker needs to be both full time AND to be an employee in the Production department, (F5 = “F” AND B5 = “Production”). In Excel, AND and OR are functions and so the way the condition is written is different from the way stated above. The word AND or OR precedes the two possible conditions, rather than appearing between them. Thus, the previous compound condition would be written as

`AND(F5=“F”,B5=“Production”)`



and the *IF* function for this scenario would look like this:

=IF(AND(F5="F",B5="Production"),G5\*0.1, "no")

The equivalent logic written as a Visual Basic statement would be:

```
If (F5 = "F" AND B5 = "Production") then
    Print G5 * 0.1
Else
    Print "no"
End If
```

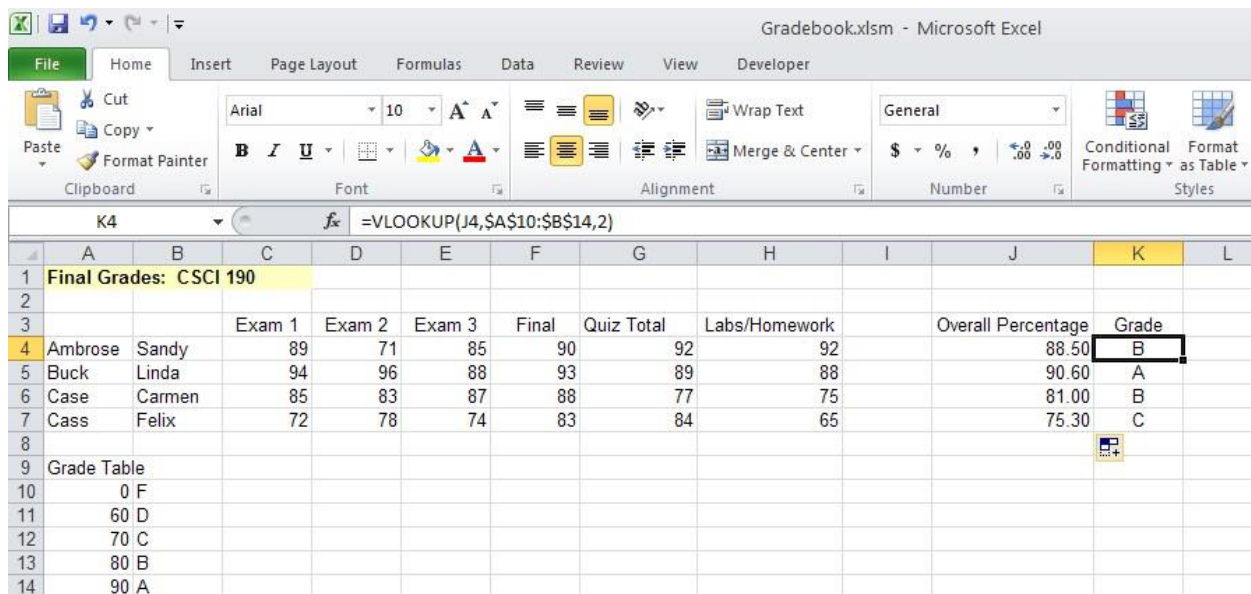
As another example, suppose we wished to give a \$500 bonus to every employee in Management or Production and a \$300 bonus to those in other departments. We could use column I for this bonus and enter in I5:

=IF(OR(B5 = "Management",B5 = "Production"), 500, 300)

Rather than typing in the function, you can also simply enter =, then choose the IF function from the function list. You will get a window with three boxes to fill in, one for the condition and two for the results when the condition is true or false.

### *VLOOKUP Function*

Sometimes we have more than two options we wish to choose from. Let's return to our gradebook example to see this function.



	A	B	C	D	E	F	G	H	I	J	K	L
1	Final Grades: CSCI 190											
2												
3			Exam 1	Exam 2	Exam 3	Final	Quiz Total	Labs/Homework		Overall Percentage	Grade	
4	Ambrose	Sandy	89	71	85	90	92	92		88.50	B	
5	Buck	Linda	94	96	88	93	89	88		90.60	A	
6	Case	Carmen	85	83	87	88	77	75		81.00	B	
7	Cass	Felix	72	78	74	83	84	65		75.30	C	
8												
9	Grade Table											
10		0 F										
11		60 D										
12		70 C										
13		80 B										
14		90 A										

The VLOOKUP function has the following form:

=VLOOKUP(x,y,z)

where x is the cell containing value to lookup, y is the lookup table range (top left and bottom right corners, designated as **absolute** references), and z is the column number in the lookup table containing the result to write. The value of z would be 3 for a three column table where the desired result is found in the third column.

The function returns the value found in the table and displays it in the cell the function is in.

The VLOOKUP function in cell K4 is:

=VLOOKUP(J4, \$A\$10:\$B\$14, 2)

The first parameter is the cell containing the value to look up in the corresponding table. The second parameter gives the range (the top left and bottom right corners) of the table. Notice that we do not include the heading and that the table reference must use absolute addressing, since we will be copying this formula down the column and we do not want the references to the table to be altered. The third parameter gives the column of the table containing the result that we wish to have placed in the cell.

In our example, we are looking up a letter to correspond with the numbers in column J. For the first student we need to look up the corresponding letter for the numeric grade in J4. Our table is in cells \$A\$10:\$B\$14. The left column of the table **must** be organized from smallest to largest. The leftmost column must match the values to be looked up. This might be a direct match, where the value being looked up will always be equal to one of the values in the table, or the table might give the lower bounds of a range, as in our example, where any grade greater than or equal to 0 and less than 60 will be considered as a match to the first line in the table. The final parameter, 2, tells the function that the result to be printed will be found in the second column of the lookup table. As with all functions, you can type in the function directly or you can choose VLOOKUP from the function table and fill in the boxes given for the parameters.

There are many more functions available in Excel. This gives you a few of the most frequently used ones. Explore the Function Library on the Formulas ribbon. Each function on this table will give you a pop up window with directions on what to enter in each of the boxes, helping to make Excel functions extremely easy to learn and use.

## 5.5 *Charts/Graphs and Templates*

### **Charts or Graphs**

Charts (or Graphs) are easy to create in Excel using the Charts group on the Insert ribbon from the menu. Highlight the data labels and then the data you wish to have on your chart. Suppose we wished to compare salaries at Gargoyle Computing, displaying the employee's names and salaries. We would first highlight the data in column A (employee names) on our Employees

worksheet, then while holding down the Ctrl key, highlight the data in column G (salaries) and then click on the desired chart type from the Charts group on the Insert ribbon.

Once you have chosen the chart type, you will see a preview of the chart with your data to see if that is the type of chart you want. If it isn't you can choose Change Chart Type from the Type group on the Design ribbon under Chart Tools. If you don't see the Chart Tools tab toward the right end of your menus, click on your chart and that will cause the Chart Tools menu to appear. Next you can enhance your graph with titles, gridlines, and data labels using choices from the Add Chart Element found in the Chart Layouts group on the Design ribbon under Chart Tools or you could instead click on the "+" sign to the right of your graph and that will bring up a list of chart elements you can select from. Finally, determine where you want the graph placed on your data sheet, or on a separate sheet in your workbook and use the Move Chart icon in the Location group on the Design ribbon to make the placement or cut and paste it where desired.

Once your graph is stored in your workbook you can enhance it in various ways. If you click on a data segment of the chart, the menu structure will change and display a Chart Tools contextual tab with the relevant menus to make changes to the chart. This allows you to easily make changes to the chart type, chart style, add new labels, or add new options to the chart. You can resize the entire chart by clicking on the border and dragging.

## **Templates**

A template is a workbook that contains formatting, formulas, and labels, basically everything that might allow a user to get helpful results with the entry of minimal amounts of data. Templates can be thought of as forms and are useful in any situation where a workbook is used repetitively for different sets of data.

Excel comes with a wide variety of built-in templates that can be customized for use. When you choose File -- New you can choose one of many templates from the Available Templates or online at Office.com. These templates give you forms that you can customize for your own use.

You can also create your own template. The easiest way to do this is to first create a workbook to use as a model for the application you desire. Format your workbook and fill in all relevant formulas but remove data that is particular to any one usage of the template. Save your template by choosing File -- Save As – and choosing Excel Template(\*.xltx) under the Save as type: drop-down menu. This will place your file in the list of templates available to you in the future.

## **5.6    *Macros***

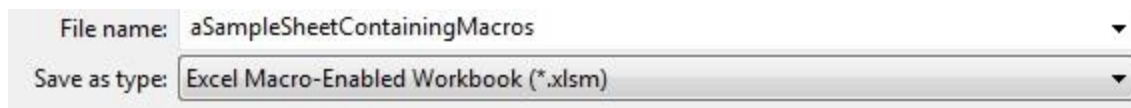
Macros allow the user to define new functions. A macro is basically a series of Excel actions that are bound together and executed on a single command. Thus with a macro one key or button on the spreadsheet accomplishes what would normally be done with a series of keystrokes and commands.

Macros are good for operations that will always be combined in a single workbook. They allow the user to easily execute the combination and ensure that the same combination of actions will be executed each time, making the workbook more foolproof. Macros are also used for operations that are done repeatedly in a single workbook. This includes operations that need to be done to entire columns or rows. Macros are also useful for templates because they allow the creator of the template to combine operations, making the template much simpler for the end user. Finally, macros can be stored outside of a particular workbook so that functions can be created that are accessible to all of a user's workbooks.

While working with recorded macros and using Visual Basic within Excel, it is helpful to have the Developer ribbon available that may not be currently visible on the main menu. To add the Developer ribbon to the menu, use the menu and click File, -> Options->, Customize Ribbon, and then select Developer from the list of Main Tabs by checking the box next to it and then click OK.

### **Creating and Saving a Spreadsheet Containing a Macro**

There are two ways to create a macro. A macro can be recorded. In this process, you simply run through a series of Excel keystrokes, and these keystrokes are recorded into a macro. A second approach is to write the macro as a VBA (Visual Basic for Applications) program. This allows you to design macros that go beyond the functions and formatting built into Excel. To be able to use macros in a spreadsheet, the file containing the macros needs to be saved as a "MacroEnabled Workbook", instead of simply an Excel Workbook. The extension on the file name for an Excel Macro-Enabled Workbook is .xlsm instead of .xlsx used for a non-macro workbook. To save a workbook containing macros, click the File tab on the main menu and then Save As and then choose Excel Macro-Enabled Workbook (\*.xlsm) from the "Save as type:" dropdown menu at the bottom of the window before giving the file a name. See the graphic below:



Since macros are programs that could be harmful, you may need to change the security setting to be able to run macros in a workbook. To change the security setting that will allow macros to run, use the menu and from the Developer ribbon, within the Code group, choose Macro Security and then click on "enable all macros ..." in the Macro Settings section. You now have to save your file, close it, and reopen it in order to run the macros.

### **Recorded Macros**

This is the simplest method of creating a macro. To record a macro, you start the recording, then do everything you want in the macro, then stop the recording. As the computer records your keystrokes or menu selections it converts them into VBA and builds a program. You will assign the macro to a control key; when you hit this control key anytime in the future, the associated program is executed.

Start by positioning the cursor where you want it to be before the macro begins. Once you begin recording, everything you do will be recorded, so you don't want to position your cursor after you start recording, unless you want the macro to begin in the same cell every time it is executed.

To start recording, from the Developer ribbon, within the Code group, choose Record Macro. You will get a pop-up window asking for a name, description, and control key for the macro. Give your macro a meaningful name, a short description, and assign it to a key for future execution.

After you click on ok, the window will disappear and everything you do now will be recorded as part of the macro until you click on Stop Recording in the Code group. Now enter the keystrokes you want as part of your macro, pressing the stop recording button when finished.

There is an icon in the Code group that allows you to set the addressing mode to relative addressing. If you don't change this, the default mode for recorded macros is absolute referencing. Macros designed with absolute addresses always use the same cells on a worksheet, in other words, if you wanted a macro that would print your name in cell A10 and your e-mail address in B10, you would use absolute addressing. You could write the same macro with relative addressing to print your name in whatever cell you position the cursor in, with your email address right below it. Relative addressing is much more flexible, and what you will want most of the time, so make sure this button is pressed before you start recording your macro when you wish to have your macro use relative addressing.

You might now try running your macro a few times, in different positions on the worksheet. If the macro does not work as you wish, you can edit it by choosing Macros from the Code group on the Developer tab. Highlight the macro you wish to change and click on Edit. This brings up a VB code module with the program for your macro in it. You can now edit the code.

Let's look at an example. Consider the following spreadsheet.

	A	B	C	D
1	<b>Gargoyle Computing</b>			
2	<b>Past Due Accounts</b>			
3				
4	<i>Customer Name</i>	<i>Balance</i>	<i>Overdue?</i>	<i>Penalty</i>
5	Smith, John	\$ 85.75	no	
6	Thomas, Anna	\$ 359.47	yes	
7	Wilkes, Valerie	\$ 92.35	yes	
8	Johns, Karen	\$ 78.37	no	
9	Mays, Matthew	\$ 573.22	yes	
10	Itenson, George	\$ 79.45	no	
11	Samuels, Sam	\$ 27.86	no	
12	Gregorson, Thomas	\$ 89.75	yes	
13	Atley, Amanda	\$ 875.32	yes	
14	Hokanson, James	\$ 475.98	no	
15				

This sheet lists customers of Gargoyle Computing who owe a balance. Column C lists whether that balance is overdue or not, and D will list a penalty. We will record two macros, one called *Sign\_it*, to add a preparer's signature in bold font to the bottom of the page, and a macro called *Penalty*, to figure out the penalty on each customer's balance. The penalty for overdue accounts will be equal to ten percent of the amount owed for debts over \$300, and five percent of the amount owed for smaller debts. For accounts that are not overdue, the penalty will be listed as "none".

To record the *Sign\_it* macro we will not use relative referencing because we want the name information to be displayed in a particular location on the worksheet, (below the employee data.) To create the *Sign\_it* macro, start recording, then position the cursor in the desired starting cell and type in the signature. After typing in the signature information, highlight the name and rightclick to choose bold font and then stop recording.

To record the *Penalty* macro, start recording, position the cursor in D5, and type in the required IF function, (=IF(C5="yes",IF(B5>300,0.1\*B5,0.05\*B5),"none")). After entering the function, format cell D5 as currency, (if the answer turns out to be "none" this will be ignored). Then stop recording.

To view the VB code for any macro, from the Developer ribbon, within the Code group, choose Macros, and then highlight the desired macro and click on Edit. The code for the two macros looks like:

```
Sub Sign_it()  
    '  
    ' Sign_it Macro  
    ' This macro will place the name of the person who prepared  
    ' this worksheet at the bottom of the data area.  
    ' The name of the person will be formatted to display in bold print.  
    '  
    ' Keyboard Shortcut: Ctrl+i  
    '  
    Range("A16").Select  
    ActiveCell.FormulaR1C1 = "prepared by:"  
    Range("B16").Select  
    ActiveCell.FormulaR1C1 = "John Miller"  
    Range("B16").Select  
    Selection.Font.Bold = True  
End Sub
```

---

```

Sub Penalty()
'
' Penalty Macro
' This macro will calculate and display the penalty on overdue accounts.
' The penalty for overdue accounts will be equal to ten percent of the amount owed for
' debts over $300, and five percent of the amount owed for smaller debts.
' For accounts that are not overdue, the penalty will be listed as "none".
'
' Keyboard Shortcut: Ctrl+u
'
    Range("D5").Select
    ActiveCell.FormulaR1C1 = _
        "=IF(RC[-1]=""yes"",IF(RC[-2]>300,0.1*RC[-2],0.05*RC[-2]),""none"")"
    Range("D5").Select
    Selection.Style = "Currency"
End Sub

```

In the code for the *Sign\_it* macro above, *Range("A16").Select* moves the cursor to cell A16 and makes that cell the ActiveCell. *ActiveCell.FormulaR1C1* simply refers to the cell the cursor is in. The words “prepared by:” will be entered into this cell. The steps recorded basically move the cursor from cell to cell and put the typed information in each. The last couple lines highlight the name entered and set the Font to Bold.

The second macro is much harder to read. This is one problem with recorded macros. The recorded macros often use terse VBA code, making the program hard to both read and edit. Thus, recorded macros are good for small things, such as adding prerecorded text, or executing a simple formula. But for more complicated formulas or functions, especially ones that you may want to edit in the future, it is often best to write your own code in VBA. You will learn about the RC notation used for the formula in the *Penalty* macro during one of your lab sessions, but at first glance you will probably agree that it is not immediately obvious what the formula is doing.

## Writing Macros Using VBA

Below is another macro solution to calculating the same penalty on overdue accounts, but this one is written in VBA code and is much easier to read and understand, (especially with the help of the comments.)



```

Sub PenaltyTwo()
'
' PenaltyTwo Macro
' created by writing the VBA code
' This macro computes the same penalty for overdue accounts as the recorded macro.
'
    Dim overdue As String           'declare all variables
    Dim penalty As Single, amountOwed As Single

    Range("D5").Select              'moves the cursor to the top of the penalty column
    amountOwed = ActiveCell.Offset(0, -2) 'assign the value of the balance due to a variable
    overdue = ActiveCell.Offset(0, -1)  'assigns overdue status, (yes/no), to a variable

    If overdue = "yes" Then          'determine if the amount owed is overdue
        If amountOwed > 300 Then      'Calculate the appropriate penalty, if any.
            penalty = 0.1 * amountOwed
        Else
            penalty = 0.05 * amountOwed
        End If
        ActiveCell = penalty         'put penalty value into the worksheet
        ActiveCell.Style = "Currency" 'format the penalty value as $
    Else
        ActiveCell = "none"          'account is not overdue, no penalty
    End If

End Sub

```

This code is much more intuitive to follow, with somewhat simpler statements than the recorded macro. *Range("D5").Select* simply makes the cell, D5, the current active cell for the macro. The variable, *amountOwed*, is assigned the value of the cell two columns to the left of the ActiveCell. An offset of (0,-2) refers to the cell that is two columns to the left of the active cell, but in the same row. The first number in the ordered pair designates the row offset from the ActiveCell. Positive numbers mean down and negative up, while zero means the cell is in the same row. The second number designates the column offset, with positive for right and negative for left and a column offset of zero indicating a reference to a cell in the same column as the ActiveCell. The variable, *overdue*, is used to hold the value of the cell one column to the left of the ActiveCell. This is the Overdue column of the spreadsheet. If this column contains "yes" then we need to calculate a penalty as a percentage of the amount owed or else we just set the penalty to zero. The calculation of the penalty is in the indented IF section. Here we determine how much the penalty is going to be, depending on the amount that is owed. After the If statements, the ActiveCell is assigned the value of the penalty and then the ActiveCell is formatted to display as currency.

This VBA code uses variables as found in most programming languages, but also uses references to spreadsheet cells using the concept of an "active" cell. When writing spreadsheet macros, we need to know where things are in relation to the ActiveCell so that we can use the Offset concept to refer to other cells in the spreadsheet. Note that we can use variables in these programs to hold the value of some cell in the worksheet as *amountOwed* was used above or to hold a value that will be put into the worksheet as *penalty* was used. Variables can be assigned a value from a cell on the worksheet or a cell on the worksheet can be assigned a value of a variable.

The PenaltyTwo macro, as written above, will only calculate the penalty for a single row in the spreadsheet. We can expand this macro to calculate the penalty for all non-empty rows on the spreadsheet by adding a Do While loop.

```

Sub PenaltyThree()
'
' PenaltyThree Macro
' created by writing the VBA code
' This macro computes the same penalty for overdue accounts as the recorded macro.
' This macro uses a Do While Loop to compute the penalty for all of the accounts.
'
    Dim row As Integer, overdue As String 'declare all variables used
    Dim amountOwed As Single, penalty As Single

    row = 0 'used with Offset to keep track of which row is being processed

    Range("A5").Select 'moves the cursor to the top of the Customer Name column

    'loop until an empty cell is found in the Customer Name column
    '(repeat while the current cell being referred to is not empty)
    Do While ActiveCell.Offset(row, 0) <> ""

        amountOwed = ActiveCell.Offset(row, 1) 'assign the value of the balance due to a variable
        overdue = ActiveCell.Offset(row, 2) 'assigns overdue status, (yes/no), to a variable

        If overdue = "yes" Then 'determine if the amount owed is overdue
            If amountOwed > 300 Then 'Calculate the appropriate penalty, if any.
                penalty = 0.1 * amountOwed
            Else
                penalty = 0.05 * amountOwed
            End If

            ActiveCell.Offset(row, 3) = penalty 'put penalty value into the worksheet
            ActiveCell.Offset(row, 3).Style = "Currency" 'format the penalty value as $
        Else
            ActiveCell.Offset(row, 3) = "none" 'account is not overdue, no penalty
        End If

        row = row + 1 'refer to the next row
    Loop
End Sub

```

This macro will compute the penalty for each person, stopping when it finds an empty cell in the Customer Name column. To begin the macro, we place the cursor in the Customer Name column rather than the Penalty column because the Penalty column is initially empty and we could not use an empty cell as the condition for terminating our loop if we started in this column.

The first thing done inside the loop is that two variables are assigned values. *amountOwed* and *overdue* are assigned the values from their respective cells in the Balance and Overdue columns. Giving these cells variable names makes the rest of the program simpler to write and read. Instead of typing `ActiveCell.Offset(row, 1)` over and over we now only have to type *amountOwed*.

To get input from the user while running a macro and place it in a cell on a spreadsheet you can use input box function:

```
ActiveCell = InputBox("Enter your name", "Namebox")
```

Data can also be read from a data file onto a worksheet by opening the file and using a loop to assign the data to a cell and then changing which cell is being referred to by incrementing the row or column offset variable, (similar to the way the variable *row* was used in the *PenaltyThree* macro.) Here is a sample VBA macro that reads data from a file and puts it in a spreadsheet:

---

```

Sub ReadDataFile()
'
' ReadDataFile Macro
' This macro will read a file containing snowfall data
' and place the data in the spreadsheet.
' Each line in the file contains two items, a city name and snow depth.

'declare variables
Dim row As Integer, city As String, snow As Single

'initialize variables
row = 0

'open data file to be read
Open ActiveWorkbook.Path & "\snowdepth.txt" For Input As #1

'place cursor in desired position
Range("A2").Select

'read the data file
Do While Not EOF(1)
    'get the data from the file
    Input #1, snow, city

    'put data in the spreadsheet
    ActiveCell.Offset(row, 0) = city
    ActiveCell.Offset(row, 1) = snow

    'increment row counter to refer to next row
    row = row + 1
Loop

'close the data file before exiting the program
Close (1)
End Sub

```

If you have studied Visual Basic, there are a few differences in the syntax between VBA and VB code that you should take note of.

1. Using VBA, you do not initialize the variables as part of the Dim statement. You must do it separately as you see with the line “row = 0”.

In VBA:      Dim row as Integer  
                  row = 0

In VB:          Dim row as Integer = 0

2. The code for opening a file is different. Note the inclusion of the pound sign, #, as part of the channel designation.

In VBA:    Open ActiveWorkbook.Path & "\snowdepth.txt" For Input As #1

In VB:      FileOpen(1, "snowdepth.txt", OpenMode.Input)

The pieces are all there in VBA, (filename, open mode, and channel number), but in a different form. Using *ActiveWorkbook.Path* as part of the file designation is similar to putting your data files in the bin/Debug folder in VB. *ActiveWorkbook.Path* refers to the location where the spreadsheet is stored. We use *ActiveWorkbook.Path* instead of the complete path to the data file including the drive letter and folder list, (e.g., M:\CS130\myExcelStuff\snowdepth.txt”), to make it easier to move your Excel project to a new location and not have to edit the path where the file is found. Note the inclusion of the backslash character, ‘\’, as part of the filename designation, “\snowdepth.txt”. That is a necessary separator between the folder name and file name in the path.

3. The use of Input is slightly different. The # sign is part of the channel designation and multiple variables can be read using one line.

In VBA:        Input #1, snow, city

In VB:        Input (1, snow)  
              Input (1, city)

4. The statement to close a file is also slightly different:

In VBA:        Close(1)

In VB:        FileClose(1)

For a summary of these VBA commands and structures and a few other tips, see Appendix B.

To begin writing your own macro using the menu system, choose Developer → Macros → (enter a new name for your macro) → Create. You are now ready to start writing the VBA code.

Another approach to create a new macro where you are prompted for the macro name, description, storage location, and shortcut control character is to begin as if you were going to record a macro. Start recording but, instead of entering keystrokes, immediately stop recording, then choose Developer → Macros → (highlight the desired macro) → Edit, to open Visual Basic and begin writing your code, (or this is how you could get to an existing macro to edit it).

### **Assigning Macros to Buttons**

So far we have assigned our macros to control keys for execution. Sometimes it is handy to have a button that is right on the worksheet that can be clicked to run the macro.

To create a button, choose Shapes from the Illustrations group on the Insert Ribbon and pick one of the shapes that will be used as a button to run your macro. Once you make your choice of shape by clicking on the desired shape, move your mouse over to the desired location on the worksheet and place the shape on the worksheet by holding down the left mouse button and dragging the mouse until the shape has the desired size. Once you release the left mouse button,

you can assign a macro to the button you just created by using a right-click of your mouse while over the shape. A menu will pop up and you can choose “Assign macro” by clicking on that choice. A list of macros will appear and you can highlight the one you wish to assign to the button and then click OK. You can put a caption on your button by moving the mouse over the button and right-clicking again, only this time choose “Edit text” from the menu. Clicking on this button will now execute the associated macro. If you wish to make other changes to the button, such as changing the color, there are several options you can explore on the popup menu when you right-click the button.

### **Global Macros**

So far, all the macros we have written have been associated with a single workbook. However, some macros might be useful to have on hand and use with more than one workbook. You can store a macro so that it is available any time Excel is running and thus can be used with any workbook in your account. To do this, when you first create the macro save it in Personal Macro Workbook, rather than This Workbook. Your personal macro workbook is open every time you run Excel and you can access macros in this workbook through their control key. To edit a macro in your personal macro workbook, choose View -- Window -- Unhide -- Personal.xls. Un-hiding the Personal workbook allows the macros within it to be edited like any other macro.

## **5.7 Example VBA Macros**

VBA Macros in Excel are especially good for using the contents of cells in a spreadsheet in a calculation without moving the cursor to those cells. Each of the following examples shows some ways to reference or select spreadsheet cells and process information in single cells, rows, columns, and blocks. In each of the code fragments below, it is assumed that variables used (if any) have been declared previously. The fragments are just to show how to do the appropriate actions.

### **Processing Data in Rows or Columns without Moving the Cursor**

For example, to compute the sum of the values in column B, beginning with row 3, until you encounter a blank cell:

```
Range("B3").Select
sum=0
row=0
Do While ActiveCell.Offset(row,0) <> ""
    sum = sum+ActiveCell.Offset(row,0)
    row = row +1
Loop
```

The style of looping over a collection of data and using a blank cell at the end of the data as a way to end the loop illustrated in the example above is a common and convenient way to structure a macro that is processing a collection of data organized in columns or rows and this method will be used often in this course.

Suppose you wanted to process a row of data starting in row 3, column B by multiplying the value in each cell by 5 and you knew that you would be done when you reached a column that was blank.

```
Range("B3").Select
column=0
Do While ActiveCell.Offset(0,column) <> ""
    'Process the element at ActiveCell.Offset(0,column)
    ActiveCell.Offset(0,column) = ActiveCell.Offset(0,column) * 5
    column=column+1
Loop
```

To go across a row of, for example, 5 items, and assuming the active cell is in the first of those five cells, (the leftmost cell), you could find the sum of the five cells using a For/Next loop. (or you could do some other desired action using these ideas)

```
For column=0 to 4
    sum = sum + ActiveCell.Offset(0,column)
Next column
```

Here is another example, to set each of 10 cells in row 2 to the value 80, starting with column C:

```
Range("C2").Select
For column = 0 to 9
    ActiveCell.Offset(0,column) = 80
Next column
```

To go down a column (say column F) of height 6, starting at row 3 and setting each cell to "A":

```
Range("F3").Select
For row=0 to 5
    ActiveCell.Offset(row,0) = "A"
Next row
```

### **Multiple Worksheets in Macros**

Often you need to use more than one worksheet. To access sheets other than the active sheet, you will need to declare a variable for each worksheet, using the Set statement to assign the variable to a particular sheet. For example, in the following statements we will set variable WS1 to sheet 'SalesSheet' and WS2 to sheet 'InventorySheet':

```
Dim WS1 As Worksheet, WS2 As Worksheet
Set WS1 = ActiveWorkbook.Worksheets("SalesSheet")
Set WS2 = ActiveWorkbook.Worksheets("InventorySheet")
```

Having variables that refer to worksheets, we can now refer to cells on the different worksheets by using the name of the sheet along with the Range name to access those cells. Here is an example which illustrates how to refer to locations on the two worksheets from within a macro. This sample code simply calculates the product of two cells, one from each of the two sheets and puts the result in the current active cell. To make a particular worksheet become the current ActiveWorksheet, use the Activate method with the desired worksheet. (e.g., WS2.Activate).

```
Dim myProduct as Single
Dim WS1 As Worksheet, WS2 As Worksheet
Set WS1 = ActiveWorkbook.Worksheets("SalesSheet")
Set WS2 = ActiveWorkbook.Worksheets("InventorySheet")
WS2.Activate           'this makes "InventorySheet" the ActiveWorksheet

Range("D5").Select     'move the cursor to this location on the ActiveWorksheet

ActiveCell = WS1.Range("C2") * WS2.Range("B4")
```

You can also specify worksheets using numbers - that is, the first (leftmost) worksheet is worksheet 1 which is referred to using Sheets(1), the second is worksheet 2, the third is worksheet 3, etc. You can use this to assign variables to individual worksheets using the Sheets method. For example, to reference the third worksheet, do:

```
Dim WorkSheet3 as Worksheet
Set WorkSheet3=Sheets(3)
```

### **Example VBA Macros**

You can write VBA macros that solve a wide variety of problems involving the data in your spreadsheet and may also involve user interaction of input boxes and message boxes to get or convey results to the user. Here are two longer example macros that illustrate some of these possibilities.

The first sample VBA macro demonstrates reading payroll data from a file into a worksheet and also introduces the concept of writing a formula for your worksheet as part of the macro. The formula refers to the desired cells on the worksheet using RC notation. RC notation is similar to the *.Offset* concept used with the ActiveCell. The code has many comments to assist you in understanding some of the new syntax used in VBA. Here is what this sample worksheet looks like after the macro was run.



	A	B	C	D
1	Name	Hours	Hourly Pay Rate	Gross Pay
2	Ed	30	\$ 14.85	\$ 445.50
3	Ned	40	\$ 12.50	\$ 500.00
4	Ted	50	\$ 10.00	\$ 500.00
5	Art	60	\$ 21.25	\$ 1,275.00
6	Sara	50	\$ 20.88	\$ 1,044.00
7	Mary	45	\$ 33.60	\$ 1,512.00
8	Martin	30	\$ 42.00	\$ 1,260.00
9	Edith	60	\$ 18.40	\$ 1,104.00

For ease of reading and studying, the entire macro is listed on the next page.

```

Sub ReadDataFile()
' ReadDataFile Macro
'This macro reads from a data file containing payroll information,
'(employee names , hours worked , and hourly pay rates),
'and inserts that data into the worksheet.
'It also creates a formula to calculate the Gross Pay for
'each person and inserts that formula into the worksheet.
Dim Row As Integer, Name As String, Hours As Single, Rate As Single
Row = 0

'open the data file on channel #1
Open ActiveWorkbook.Path & "\payroll.txt" For Input As #1

'position the cursor in cell A2, (making it the ActiveCell)
Range("A2").Select

'read all of the data from the file, (While Not End Of File)
Do While Not EOF(1)
    'read the first three items from the data file
    Input #1, Name, Hours, Rate

    'put the data into cells in the worksheet
    ActiveCell.Offset(Row, 0) = Name
    ActiveCell.Offset(Row, 1) = Hours
    ActiveCell.Offset(Row, 2) = Rate

    'format the hourly rate to display as $   xxx.xx
    ActiveCell.Offset(Row, 2).Style = "Currency"

    'create a formula to calculate the Gross Pay
    ' The RC notation below is creating a formula to multiply
    ' the two cells just to the left of the Gross Pay column,
    ' (The Hours and the Hourly Pay Rate.)
    ' The RC notation uses the idea of offset in the same way
    ' that it is used with Activecell.Offset(Row,Col)
    ' R[0]C[-2] means the cell two columns to the left,
    ' in the same row as the formula.
    ActiveCell.Offset(Row, 3).Formula = "=R[0]C[-1]*R[0]C[-2]"

    'format the Gross Pay as currency
    ActiveCell.Offset(Row, 3).Style = "Currency"

    'increment Row to refer to the next employee
    Row = Row + 1
Loop

'close the data file
Close (1)
End Sub

```

---

The second sample macro: Your boss has a new idea to improve employee morale. She already has a special award for the top salesperson each month. She thinks this is unfair because the same people seem to win the award every month. She will stop giving the ‘top’ award and, instead,

give an award to the salesperson who finishes 5th. She wants you to design a macro for her salesperson spreadsheet that will find the 5th highest salesperson and display that name in a message box. Her salespeople are listed in column 1 and their total sales in column 2, starting in row 4. The number of salespeople varies, but you may assume the row after the last salesperson has an empty cell in column 1 and there are no more than 500 salespeople working for the company. You may also assume you are working in the active worksheet.

Solution: Whenever you want to find a particular 'rank' in a column or row, the thing you normally do first is to copy the values in the column or row into an array and sort the array. In our particular case, we will copy both the first column and the second column into arrays, sort them together, and, then, put the 5th from the top salesperson's name into a message box.

For ease of reading and studying, the entire macro is listed on the next page.

```

Sub FifthHighest_()
    ' this macro will find and display the name of the salesperson
    ' who has the fifth highest sales amount in the list.

    'declare the variables needed
    Dim salesPerson(500) As String, sales(500) As Double
    Dim numElements As Integer, row As Integer
    Dim tempSales As Single, tempSalesPerson As String

    'initialize variables
    numElements = 0
    row = 0

    'move the cursor to the desired starting location
    Range("A4").Select

    'Read the data from the spreadsheet into two arrays
    Do While ActiveCell.Offset(row, 0) <> ""

        'put the names and sales amounts in the two arrays
        salesPerson(numElements) = ActiveCell.Offset(row, 0)
        sales(numElements) = ActiveCell.Offset(row, 1)

        'numElements is used as the location within the arrays and as a counter
        numElements = numElements + 1

        ' increment row offset to refer to the next name
        row = row + 1
    Loop

    'the following code uses the Bubble Sort to sort the sales array into descending order
    'and since there are parallel arrays in use, whenever two of the sales items
    ' are swapped, the corresponding salesPersons are also swapped).

    For pass = 0 To numElements - 2
        For Position = 0 To numElements - 2 - pass
            If sales(Position) < sales(Position + 1) Then 'swap if necessary
                tempSales = sales(Position)
                sales(Position) = sales(Position + 1)
                sales(Position + 1) = tempSales
                tempSalesPerson = salesPerson(Position)
                salesPerson(Position) = salesPerson(Position + 1)
                salesPerson(Position + 1) = tempSalesPerson
            End If
        Next Position
    Next pass

    MsgBox (salesPerson(4) & " is the Fifth best salesperson!")
End Sub

```

## **5.8 Conclusion**

Spreadsheets are extremely useful for any application that uses large amounts of numeric data. Spreadsheets allow you to store, organize, and process data. Output can be in tabular or graphic form. Users can perform a variety of operations on their data through the use of built-in functions or through macros designed by the user. This gives Excel all the processing power of Visual Basic.

# Chapter 6

## *How Computers Work: Data Representation*

The underlying purpose of a computer is to process data. We can imagine a stream of data which passes into the computer, is modified in some way, and then passes out of the computer. A piece of data, moving from one place to another, is what we mean by a signal. Signals are the basis of all communication.

### **6.1    *Analog and Digital Signals***

Consider a few common signals in daily life. Red, yellow, and green traffic control signals are an obvious example. The late Roger Ebert and Gene Siskel made “thumbs up” and “thumbs down” signals for their opinion of a movie's quality. The numbers displayed on scoreboards in stadiums and arenas are signals of the progress of an athletic event. Other signals include the speedometer needle in a car which indicates the car's speed, the level of a column of mercury in a thermometer which indicates a temperature, the position of the arrow on a bathroom scale, indicating your weight.

#### **Analog signals**

The last three are examples of *analog signals*. An analog signal is a signal which can take on a continuous range of values which are analogous to the thing they measure: a speedometer typically measures any value from 0 to 220 kilometers per hour (or 0 to about 140 miles per hour for those who still haven't gone metric); a thermometer will measure some range of temperatures between the freezing point ( $-38.87^{\circ}\text{C}$ ) and boiling point ( $356.9^{\circ}\text{C}$ ) of mercury; a bathroom scale can indicate any weight between 0 and 300 pounds. The common characteristic of these analog signals is that, within the range they measure, the values they can take on are continuous. That is, between any two analog signal values, no matter how close together, there are more possible values, at least in principle.

The difficulty with analog signals is in measuring them precisely. Although analog signals can measure a continuous set of values, in principle; in practice, the more finely we wish to distinguish signal values, the more difficult they are to measure. It is necessary to consider the amount of precision that can be used accurately in making measurements. For example, when standing on an analog bathroom scale, you see the arrow pointing to your weight, but you can probably only tell that weight to the nearest pound.

#### **Digital signals**

There is an alternative to analog signals. The first three examples at the beginning of the chapter are digital signals. A digital signal is composed of some finite set of discrete values: the traffic signal can take on three discrete values, (red, yellow, and green), and there are no intermediate values; Siskel and Ebert limited themselves to just two possible judgments about a movie, either

“thumbs up” or “thumbs down”, although they often tried to qualify things a bit with phrases like “marginal thumbs down” or “two thumbs, way up;” and of course, the scores on a scoreboard are represented as two or three decimal digits. The characteristic of digital signals is that they take on discrete values; every value has one or more adjacent values which are distinct from the starting value but allow no intermediate values between them. This means that the limits to the precision of digital values are built into the representation scheme. This limitation is a “feature not a bug” because it allows us to have great confidence in the accuracy of any measurement. We can tell if a traffic signal is green or not, and when it changes to yellow, we notice immediately and make a quick decision about whether to stop. Imagine an analog traffic signal that gradually changed from green to yellow and then just as gradually turned to red! It would be quite difficult for drivers to determine what they were supposed to do, if they noticed the change at all.

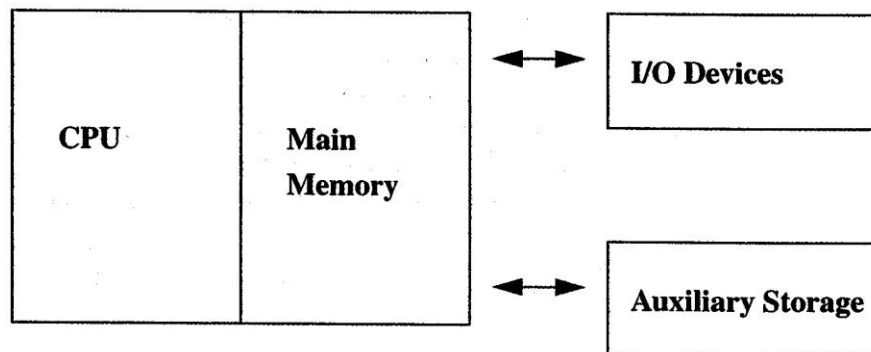
The big advantage of digital signals, then, is the relative ease of measuring them and the consequent accuracy of the measurement. Furthermore, we can compose digital signals of smaller digital components to allow us to limit the effective range of measurements. This is the principle behind the Arabic numeral system that we use today for numbers. Decimal Arabic numerals are composed of only ten fundamental signals, the digits ‘0’ to ‘9’. Considering only the natural numbers for the moment, we need only one digit to represent the numbers 0 to 9, two digits for the numbers 0 to 99, three digits for 0 to 999, etc.

Digital signals are the most appropriate way to represent data in computers. Before we look more closely at what type of digital coding system to use, let's look briefly at how a signal might be stored in a computer. This will help us understand why computers use the encoding schemes that they do.

## 6.2 Overview of Computer Design

Computer hardware falls into four basic categories: processors, memory, input and output (I/O) devices, and auxiliary storage devices. Every computer must have a processor, known as the central processing unit, or CPU, and a main memory. Most computers also have the capacity for input and output of some sort and for long-term storage. However, since these last two are not necessary in order to have a functioning computer we often call I/O and auxiliary memory devices

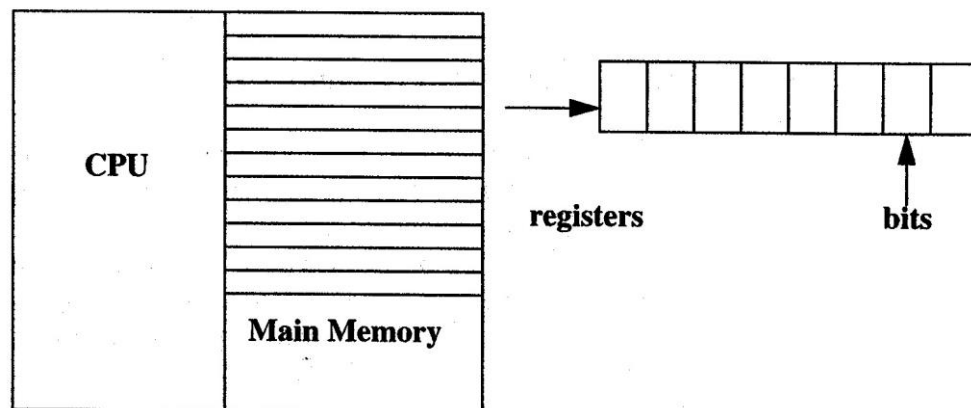
since it  
to the  
of the



peripheral  
equipment,  
is peripheral  
basic  
functioning  
computer.



Data is entered into the computer through some input device, perhaps a keyboard. This data is stored in main memory. Main memory represents a holding ground for the data both before and after it is acted upon in the CPU, or central processing unit. The CPU contains the circuits that act upon the data, with the result being stored again in main memory. To see the result, it is transferred to some output device such as a printer or a monitor. If the result is to be stored for some period of time, it is transferred to some form of auxiliary storage, such as a hard disk or CD.



Data that is entered into a computer from some input device needs to be transformed into a form that can be stored in the computer's main memory. This memory is made up of a large bank of *registers*. Think of a register as a storage slot that can hold one or more pieces of data, perhaps a number or a character or two from the keyboard. Each individual register is identified by an address. We can think of registers as if they were the boxes in a post office. Each is identified by a number, starting with 0, and each can hold one or more objects as contents.

Address

Contents

The contents segment of a register is made up of *bits* or *binary circuits*. These circuits are called binary because each one can assume two states, on or off. Think of a circuit as being like a light switch. When it is in the on position there is current going through it and when it is off there is no (or only a very little) current going through it. We will represent these two states with a 1 for on and a 0 for off. Addresses are represented the same way, though they are not physically stored in the computer. Thus we can represent the address and contents of a register as series of 0s and 1s.

The number of bits in a register varies from computer to computer. Bits are grouped together in larger units called *words* which represent the smallest unit that would be used to store a piece of data on that computer. Typically, the smallest addressable number of bits that would allow us to store a number or a character is eight, and an eight-bit word is known as one *byte*. Typical word sizes are sixteen, thirty-two, and sixty-four. Typical register sizes are sixteen, thirty-two, and sixty-four. The earliest personal computers had eight-bit registers. While few computers on today's market have registers as small as one byte, we will use this size for illustration in this text, to make things simpler.

Small special purpose computers may have only several hundred registers in main memory. Most general purpose computers have millions or even billions of registers. The size of a computer's main memory is measured by the number of bytes in that memory rather than the number of registers. These bytes are generally counted in groups of either 1024 (roughly 1000, thus one *kilobyte*, or KB) or 1,048,576 (roughly one million, thus one *megabyte*, MB). We use 1024 and 1,048,576 rather than 1000 and 1,000,000 because  $1024 = 2^{10}$  and  $1,048,576 = 2^{20}$ .  $2^{30}$  is known as a *gigabyte* (GB). You will see in the next section why we use powers of two to build our memory.

We need methods for converting any data we wish to store in the computer's memory into sequences of 0s and 1s. The data stored in computers includes integers, real numbers, characters, pictures, and sound so we will need a conversion method for each of these. Since the address side of each register needs to hold a positive integer, let's begin by looking at how integers might be stored.

### 6.3 *Binary Representation of Positive Integers*

The number system that we use every day is called the decimal system. The word decimal means that it is based on powers of ten. A number, such as 253 is equal to 3 ones, 5 tens, and 2 hundreds.

Each place in a decimal number stands for a power of 10:

...	10000s	1000s	100s	10s	1s
...	$10^4$	$10^3$	$10^2$	$10^1$	$10^0$

Note that we also have 10 possible symbols that can stand in each place, the digits 0 through 9. For our registers we need a system with only two possible symbols, 0 and 1. The *binary* system fits this requirement. Binary numbers are written with each place representing a power of 2 rather than 10.

...	16s	8s	4s	2s	1s
...	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$

To convert from binary to decimal, simply multiply each 1 in the binary number by the value of its place. For example, working right to left, the binary number 10011 would equal  $1*1 + 1*2 + 1*16 = 19$  in decimal. Any positive integer that we can express in decimal can likewise be converted into binary by determining the powers of two that make up that number. For example,  $37 = 32 + 4 + 1 = 100101$  in binary. With eight bits we can represent the numbers from 0 (00000000) to 255 (11111111).

Binary gives us a natural way to transform positive integers into sequences of 0s and 1s and this is how most computers store such integers. However, while binary is easy to store in a computer's memory, it is not easy for humans to read, especially when the numbers get large. Most computers today have registers of at least 16 bits, and often 32 or 64, and there are times when we might want to discuss the bit pattern in a given register. Consider the patterns:

1011010011101110 sixteen bit

11001100111100101010011000111100 thirty-two bit

It would be easy to make a mistake in transmitting either of these patterns from one user to another. It is easier for computer scientists to use a shorthand when discussing long bit patterns rather than transcribing the entire pattern. The most common shorthand is to put the number into base 16 or hexadecimal. The reason base sixteen is used is that 2 to the fourth power is sixteen and that makes it convenient to switch between base two and base sixteen, since any four binary digit pattern, (of which there are exactly sixteen), can be replaced with its equivalent base 16 digit. In hexadecimal numbers each place represents a power of sixteen, thus we have

...	4096s	256s	16s	1s
...	$16^3$	$16^2$	$16^1$	$16^0$

There is one problem. In base 2 we used two symbols in each place, 0 and 1. In base 10 we used the ten symbols of 0 through 9. In base sixteen we therefore need sixteen symbols to represent the decimal numbers 0 through 15. For this we use 0 through 9 and then the letters A, B, C, D, E, and F to represent 10, 11, 12, 13, 14, and 15. Thus the decimal number 31 would equal  $1F = 1*16 + 15$  in hexadecimal. It is common to refer to base 16 numbers as “hex”, base 2 numbers as “binary”, base 10 numbers as “decimal” and base 8 numbers as “octal”.

Since computer scientists use hexadecimal primarily to describe long binary strings, we rarely convert between hexadecimal and decimal, but generally convert between hexadecimal and binary. Fortunately, there is an easy way to make this conversion. Note that with four places in binary we can represent the decimal numbers 0 (0000) through 15 (1111). Thus if we take a binary number and divide it into groups of four digits, starting from the right, we can convert each of those groups to the corresponding number or letter in hexadecimal.

1001101010000110 = 1001 1010 1000 0110 = 9A86 in hex.  
AF20 in hex = 1010 1111 0010 0000 = 1010111100100000 in binary.

Following is a table showing the values of several numbers in all three bases:

Decimal	Binary	Hex
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	10000	10
17	10001	11
64	1000000	40
256	100000000	100

## 6.4 Rules of Binary Addition

When adding binary numbers together, the sum will be expressed as another binary number, where each digit of the sum represents a power of two.

Here is a table that summarizes the rules for binary addition:

+	0	1
0	0	1
1	1	10

Another way to say this is to say:

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$1 + 1 = 10$  (but two in binary requires that we carry the two to the next place value)

□ *Note that  $1 + 1 + 1 = 11$  in binary (1 two and 1 one, or three in decimal)*

Addition of binary integers can be carried out column by column, similar to adding decimal integers. Following is shown the binary addition of two and three along with the decimal version:

$$\begin{array}{r} 10 \\ + 11 \\ \hline 101 \end{array} \qquad \begin{array}{r} 2 \\ + 3 \\ \hline 5 \end{array}$$

Adding the numbers in the right column,  $0 + 1 = 1$  and for the left column,  $1 + 1 = 10$ . The binary number 101 represents the number five. From left to right the 101 represents one  $2^2$ , zero  $2^1$ , and one  $2^0$ , which can be interpreted as 1 four and 1 one, or five.

## 6.5 Two's Complement Representation for Positive and Negative Integers

Positive integers are sufficient for storing a memory cell's address, but for arithmetical calculations we need both positive and negative integers. The simplest method for adding negative integers to binary representation would be to designate one bit as a sign bit. The usual choice is to let the leftmost bit represent the sign, with 0 for positive and 1 for negative values. The remaining bits represent the number in binary. This method is called *sign-magnitude* representation.

$$\begin{array}{l} 10000011 = -3 \\ 01111111 = 127 \end{array}$$

There are two problems with sign-magnitude representation. First, we have two possible representations for 0, 00000000 and 10000000. This could be a problem when comparing the results of two arithmetic operations, since one might result in 00000000 and the other in 10000000 and these patterns are not the same, even though both are equal to 0.

A second difficulty occurs when we try to add two numbers in sign-magnitude. The usual rules for binary addition are  $0 + 0 = 0$ ,  $0 + 1 = 1$ , and  $1 + 1 = 10$ . However, if we were to add -3 and 12 with these rules, we would get

$$\begin{array}{r} 10000011 \quad (-3) \\ + \quad 00001100 \quad (12) \\ \hline 10001111 = -15! \end{array}$$

A better way to represent both positive and negative integers, and the method used by virtually all computers today, is with *two's complement* representation. In two's complement, we let all the places hold the value they would hold in binary, but the leftmost place is negative. Thus, for an eight bit machine we would have the following values for each place:

$$\begin{array}{cccccccc} -128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \end{array}$$

Numbers with a 0 in the leftmost slot are positive, and look like regular binary. Numbers with a 1 in the leftmost slot are negative, and their decimal value can be found by adding the values for the seven rightmost bits and then subtracting 128. Thus 11010110 would equal  $2 + 4 + 16 + 64 - 128 = -42$ .

Converting from decimal to two's complement is a little harder. For positive numbers, simply find the binary representation, making sure there is a 0 in the leftmost slot. For negative numbers, you could figure out what combination adds to the correct number, but there is an easier way. First, find the binary representation for the positive equivalent. Now flip all the bits and add 1. For example, to find -52, first find  $+52 = 00110100$ . Now flip all the bits to 11001011. Now add 1, getting  $11001100 = -52$ . To check this, note that  $-128 + 64 + 8 + 4 = -52$ .

Note that we have solved some of the problems we had in sign-magnitude notation. We now have only one representation for 0, and it is the obvious one of 00000000. We also can use the usual laws of binary addition. If we add -3 and 12 in two's complement, we get

$$\begin{array}{r}
 11111101 \quad (-3) \\
 + 00001100 \quad (12) \\
 \hline
 00001001 = 9
 \end{array}$$

Note that we carried a 1 from the leftmost addition and this carry was lost since we only had eight bits to put our answer in. Still we got the correct answer. This is because we both carried into and carried out of the leftmost column. If we carry into the leftmost column and do not carry out, we are adding two positive numbers whose sum would be too large to fit in eight bits and we get an incorrect result:

$$\begin{array}{r}
 01100000 \quad (96) \\
 + 01000001 \quad (65) \\
 \hline
 10100001 = -95!
 \end{array}$$

This condition is called *overflow*.

Similarly, if we carry out of the leftmost column, but did not carry into it we are adding two very small negative numbers, ones that would result in an answer too small to be represented in the number of bits available. Again we get an incorrect result:

$$\begin{array}{r}
 10000001 \quad (-127) \\
 + 10000110 \quad (-122) \\
 \hline
 00000111 = 7!
 \end{array}$$

This condition is called *underflow*. Since most computers have much larger registers, overflow and underflow are not very common. However, when working with very large positive integers

or very small negative integers one may need more than one register in order to avoid both overflow and underflow. Most computer systems allow the user to specify when this is the case and will use two registers to store the numbers, effectively doubling the number of bits available for representing the number. This is generally called *double precision*.

## 6.6 *Floating Point Representation for Real Numbers*

Although we can solve many problems with integers alone, many other computer applications involve the use of numbers that have a fractional or decimal part. Mathematicians call such numbers real numbers. In decimal, we represent real numbers by using a decimal point. The places after the decimal point represent the fractional powers of ten,  $1/10$ ,  $1/100$ ,  $1/1000$ , etc.

$$42.51 = 4 * 10 + 2 * 1 + 5 * 1/10 + 1 * 1/100$$

In binary, we could use a similar point (here called a radix point since the term decimal refers to base ten) and let the places to the right of this point represent fractional powers of two.

$$1001.11 = 8 + 1 + 1/2 + 1/4 = 9 \frac{3}{4}$$

One problem with using a radix point in a computer representation is that there is no easy way to represent the point in a bit pattern that uses only 0s and 1s. Today's computers generally use a method called *floating-point* representation to represent the number without needing a radix point.

Floating-point representation is similar to scientific notation. In scientific notation, a number has three parts: The first part is the *sign*: positive (+ or blank) or negative (−). The second part, called the *mantissa*, consists of a fixed number of digits with *one* non-zero digit in front of the decimal point and one or more digits after the decimal point. The third part, the *exponent*, consists of the letter *E* followed by a sign and an integer, which designates the corresponding power of ten. For example,  $4.32E3$  stands for  $4.32 * 10^3 = 4320$ . The sign is positive, the mantissa is 4.32, and the exponent is 3. Another example:  $1.55E-2 = 1.55 * 10^{-2} = 1.55 * 1/100 = .0155$ . To put a number into scientific notation, divide or multiply by 10 until you have one non-zero digit in front of the decimal point. Starting with a zero exponent, add one to the exponent for each time you divide by 10 or subtract one for each time you multiply by 10.

Floating-point representation works in much the same way, except it uses binary. We will store a sign, mantissa, and exponent for our number. For a hypothetical eight-bit machine, we can use one bit for the sign, four bits for the exponent, and three for the mantissa:

0	0000	000
sign	exponent	mantissa



This scheme will not allow us to store very large or small numbers, nor does it give us much precision, but it will serve to illustrate how floating-point representation works. Most computers today use 32, 64, or even more bits for floating-point representation, providing a wide range of numbers with high precision.

Our eight-bit scheme will use a sign with 0 for positive and 1 for negative. For our exponent, we need both positive and negative values and we have only four bits to work with. Regular binary notation would give us the positive values from 0 (000) to 15 (1111), but this doesn't work because we need to include negative exponents. We could use a two's complement system, which would give us values from -8 (1000) to 7 (0111), but it turns out that it's better to have 0000 represent the lowest exponent (even if it's not zero) and to have 1111 represent the highest exponent, with a total range of sixteen exponents. We can choose any range we want; in this case, we'll use -7 to 8. We shift each exponent down by seven, letting 000 represent -7, 001 represent -6, up to 1111 representing 8. Thus, the value represented by the four-bit exponent will be seven less than the binary equivalent for the pattern in those bits.

Exponent Pattern	Decimal Equivalent	Power of Two
0000	-7	1/128
0001	-6	1/64
0010	-5	1/32
0011	-4	1/16
0100	-3	1/8
0101	-2	1/4
0110	-1	1/2
0111	0	1

Exponent Pattern	Decimal Equivalent	Power of Two
1000	1	2
1001	2	4
1010	3	8
1011	4	16
1100	5	32
1101	6	64
1110	7	128
1111	8	256

The mantissa in scientific notation always has one non-zero digit in front of the decimal point and some fixed number of places after the decimal point. For our binary floating-point representation, we will use one binary digit in front of the radix point and three after. Since we have only 0 and 1 as symbols in binary, we will always end up with a mantissa of the form 1.xxx, and since there is always a 1 in the first position, we do not need to represent it! For this reason, we can use all three mantissa bits for the fractional part, with place values of 1/2, 1/4, and 1/8.

Suppose a register is holding a floating-point number with the bit pattern of 11100011. The first bit is the sign. Since it is a 1, the number is negative. The next four bits, 1100, are the exponent. 1100 is 12 in binary. We subtract 7 (our fixed offset) and get 5 as our exponent. This means we will multiply our mantissa by  $2^5$  or 32. The last three bits, 011 are the mantissa, without the initial 1. Since these represent halves, fourths, and eighths, respectively, our mantissa is  $1 + 1/4 + 1/8 = 1 \frac{3}{8}$ . Note that we would have gotten the same result for our mantissa if we had simply converted the pattern in the four bits to decimal, divided by eight, and added one. The final number is  $-1 \frac{3}{8} * 32$  or -44.

Now consider the bit pattern 00011000. The sign is positive since the first bit is 0. The exponent is 0011 = -4, so we will multiply the mantissa by  $2^{-4}$  or 1/16. The mantissa is 000 = 1. Thus the final number is  $+1 * 1/16$  or just 1/16.

### Algorithm for 8-bit floating-point to decimal conversion

- 1) break the bit pattern into *sign* (first bit), *exponent* (next four bits), and *mantissa* (last three bits)
- 2) *sign* = + if 0 and - if 1
- 3) *exponent* = decimal equivalent of *exponent* bits - 7
- 4) *mantissa* = 1 + decimal equivalent of *mantissa* bits / 8
- 5) *number* = *sign mantissa* \*  $2^{\text{exponent}}$

To convert a number from decimal to floating-point, we first need to get the number into the form  $\pm 1.x * 2^{\text{exp}}$ . To do this we will have to either multiply or divide by two until we have a 1 in front of the decimal point. Each time we multiply by two, we subtract one from the exponent and each time we divide by two, we add one to the exponent. We then convert both the exponent and the mantissa into binary.

For example, given the decimal number -4.44, we first set the sign bit to 1. We now need to transform 4.44 to a number of the form  $1.x$ . We divide by two until we have this form, adding one to our exponent each time.

exponent	mantissa
0	4.44
1	2.22
2	1.11

We now add the offset 7 to the exponent, getting 9 or 1001 in binary. We drop the 1 from the mantissa and write the fractional part in terms of eighths, getting  $.11 * 8/8 = .88/8$  or approximately 1/8, written as 001 in binary. Our final floating-point number is 1 1001 001.

Now consider the decimal number 0.34. We set the sign bit to 0 since the number is positive. Next we transform 0.34 to the form  $1.x$  by multiplying by two, subtracting from the exponent each time.

exponent	mantissa
0	0.34
-1	0.68
-2	1.36

Adding 7 to the exponent gives us 5 or 0101 in binary. We drop the 1 from the mantissa and write the rest in terms of eighths, getting  $.36 * 8/8 = 2.88/8$  or approximately 3/8, written as 011 in binary. The final floating-point representation is 0 0101 011.

### Algorithm for decimal to 8-bit floating-point conversion

- 1) set the sign bit to 0 if the decimal *number* is positive or 1 if negative
- 2) start with the *exponent* = 0, *mantissa* = absolute value of the *number*

- 3) while  $mantissa < 1$  do  
 $mantissa = mantissa * 2$   
 $exponent = exponent - 1$   
while  $mantissa \geq 2$  do  
 $mantissa = mantissa / 2$   
 $exponent = exponent + 1$
- 4) if  $exponent < -7$  or  $exponent > 8$  then number is outside allowable range
- 5)  $exponent = exponent + 7$ , convert into 4-bit binary integer
- 6)  $mantissa = (mantissa - 1) * 8$ ,  
round to the nearest integer and convert into 3-bit binary integer
- 7)  $number = sign\ exponent\ mantissa$

One difficulty with this floating-point method is that there is no representation for 0. This occurs because we always assume a hidden 1 in the mantissa. Thus the floating-point pattern 00000000 which we would expect to be 0 actually represents  $1 * 2^{-7}$  (after the exponent shift of 7) or 1/128! The standard solution treats an exponent of  $-7$  (0000) as a special case in a way that allows this bit pattern to represent zero.

A second difficulty that you have probably already noticed is that in converting from decimal to floating-point we round off the mantissa before putting it into binary. This means that our floating-point representation will often not be exact. Of course, part of the problem lies in the fact that we have been using an eight-bit register, which is much too small for practical computation. For greater accuracy in calculating with floating-point, we need 32 bits or more. Even then, some numbers require rounding off, but this is not a new problem when working with real numbers. Recall that even in our decimal system we have numbers that we must round off for practical calculations, such as  $\pi$  (3.14159...) or  $1/3$  (0.333...). Even so, the user should be aware that very large or very small integers and many real numbers have only an approximate representation and some have no representation at all in any given system. Calculations that require extreme accuracy or range may require special representation systems and/or a computer with a very large register size.

## 6.7 Representing Text and Symbols

Of all the different kinds of data stored and processed by computers, much of it is text data. Humans communicate most through language, so we need a method for transforming text into patterns of 0s and 1s.

A line of text can be broken down into the characters that comprise it. There are character sets for all different alphabets. Here we will look at how to encode the Latin alphabet that is used for the English language.

There is no obvious way to associate a character with a bit pattern, as there was for numbers. Thus the choice of a code to transform letters and symbols into patterns of 0s and 1s is entirely

arbitrary. There are several such codes, however only a few have become standardized. One of the earliest standards was the American Standard Code for Information Interchange (ASCII) which was developed by the American National Standards Institute (ANSI) and has been used in a variety of forms throughout the USA and Europe. All personal computers use some form of the ASCII code. Standard ASCII is a seven bit code for 128 different characters, including the 26 upper case letters A through Z, 26 lower case letters a through z, 10 digits 0 through 9, 33 symbolic characters such as #, !, and the blank space, and 33 control characters, such as a carriage return or a tab. More recently another standard, UNICODE, has become more widely used as it includes codes to represent many of the world's alphabets and writing systems. UNICODE has many more characters and symbols available and therefore requires many more bits per character than the 7-bit ASCII code. There are several versions of UNICODE that require from 8 to 48 bits per character.

The following table shows a listing of all 128 of the codes in the original ASCII code.

**ASCII Code Table:**

Ctrl	Dec	Hex	Char	Code	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
^@	0	00		NUL	32	20	!	64	40	@	96	60	`
^A	1	01		SOH	33	21	!	65	41	A	97	61	a
^B	2	02		STX	34	22	"	66	42	B	98	62	b
^C	3	03		ETX	35	23	#	67	43	C	99	63	c
^D	4	04		EOT	36	24	\$	68	44	D	100	64	d
^E	5	05		ENQ	37	25	%	69	45	E	101	65	e
^F	6	06		ACK	38	26	&	70	46	F	102	66	f
^G	7	07		BEL	39	27	'	71	47	G	103	67	g
^H	8	08		BS	40	28	(	72	48	H	104	68	h
^I	9	09		HT	41	29	)	73	49	I	105	69	i
^J	10	0A		LF	42	2A	*	74	4A	J	106	6A	j
^K	11	0B		VT	43	2B	+	75	4B	K	107	6B	k
^L	12	0C		FF	44	2C	,	76	4C	L	108	6C	l
^M	13	0D		CR	45	2D	-	77	4D	M	109	6D	m
^N	14	0E		SO	46	2E	.	78	4E	N	110	6E	n
^O	15	0F		SI	47	2F	/	79	4F	O	111	6F	o
^P	16	10		DLE	48	30	0	80	50	P	112	70	p
^Q	17	11		DC1	49	31	1	81	51	Q	113	71	q
^R	18	12		DC2	50	32	2	82	52	R	114	72	r
^S	19	13		DC3	51	33	3	83	53	S	115	73	s
^T	20	14		DC4	52	34	4	84	54	T	116	74	t
^U	21	15		NAK	53	35	5	85	55	U	117	75	u
^V	22	16		SYN	54	36	6	86	56	V	118	76	v
^W	23	17		ETB	55	37	7	87	57	W	119	77	w
^X	24	18		CAN	56	38	8	88	58	X	120	78	x
^Y	25	19		EM	57	39	9	89	59	Y	121	79	y
^Z	26	1A		SUB	58	3A	:	90	5A	Z	122	7A	z
^[	27	1B		ESC	59	3B	;	91	5B	[	123	7B	{
^\	28	1C		FS	60	3C	<	92	5C	\	124	7C	
^]	29	1D		GS	61	3D	=	93	5D	]	125	7D	}
^^	30	1E	▲	RS	62	3E	>	94	5E	^	126	7E	~
^-	31	1F	▼	US	63	3F	?	95	5F	_	127	7F	␣*

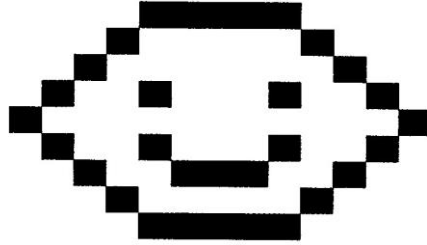
Computers with larger registers can store more than one character per register. We only need seven bits to store 128 different bit patterns, but bits generally come in bytes, or multiples of eight. This leaves an extra bit for each character. In computer networks, this bit is often used to check for errors in transmission and is called a *parity* bit. The transmitting and receiving computers agree on either an odd or even parity, which means that each byte will have either an odd or even number of ones in it. Thus, if two computers were working with odd parity and the character A were transmitted a 1 would be added to the pattern 1000001 that represents A so that the total byte would have an odd number of 1s in it, (11000001). Errors in transmission often corrupt only a single bit, thus the character would come through with the wrong parity. To correct the error, the character would have to be retransmitted, since the receiving computer has no way of knowing which bit is the faulty one. Another common use for the extra bit is to extend the character set to 256 characters, a code known as ASCII-8 that contains patterns for extra mathematical and graphical symbols and diacritical marks.

At this point, we should note two things. First, we have three different methods for representing integers: two's complement, floating-point, or encoding each digit in ASCII. How we choose to represent a number depends on the use we plan to make of that number. If we plan to calculate with the number we must store it in either two's complement or floating-point, the choice of which depends on the nature of our calculations (though most computer applications include programs that will convert a two's complement number to floating point when necessary). Numbers stored in ASCII cannot be used in calculations. However, when storing the digits of a phone number or street address it makes sense to encode them in ASCII since these numbers are unlikely to be used in a calculation and the data surrounding them is probably stored in ASCII.

Second, consider the bit pattern 00110100. This pattern could now represent the two's complement integer 52, the floating-point real number  $1.25 * 2^{-1}$  or .625, or the ASCII code (with odd parity) for the digit 4. Obviously we need some method of knowing which it is. There are internal codes that sometimes precede a block of data, telling the computer which type of data is in that block. When writing a program, you will need to specify what sort of data is stored in each variable in order to set these codes.

## **6.8    *Representing Pictures***

Pictures are stored by dividing the screen into a grid of cells called *pixels*. For a black and white picture, we can store a 1 for each cell that is black and a 0 for each cell that is white. If the grid is fine enough (has a high enough resolution) a fairly accurate representation of the picture will be stored. A normal grid would have a resolution of at least  $512 * 256$ , which is the resolution of an old black and white television set. This does not take too much storage, since we only need one bit per pixel.



An enlargement of a simple 16 x 9-pixel picture above could be stored as:

```
0000001111100000
0000010000010000
0000100000001000
0001001000100100
0010000000000010
0001001000100100
0000100111001000
0000010000010000
0000001111100000
```

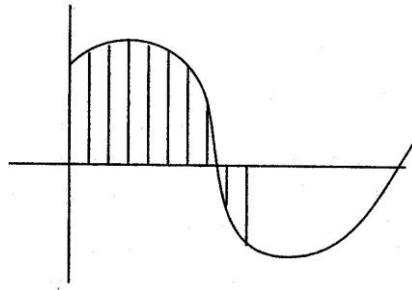
Shading is accomplished by storing a value between 0 and 255 for each pixel, with 0 representing white and 255 representing black, with the numbers in between as shades of gray. Color requires the storage of three numbers for each pixel, called RGB triples. These three numbers represent the relative amount of Red, Green, and Blue making up the color of the pixel. There are other formats for storing the color of a pixel that store information about the hue, saturation and intensity, but the RGB format is the most common.

Note the high storage cost for color pictures. Assuming each number can be stored in 1 byte, one full screen color picture would take  $512 * 256 * 3$  or 384K (kilobytes) of memory. Moving pictures take even more. Most motion pictures have around 30 frames per second. Thus you would need  $384K * 30 * 60 * 120 = 83$  billion bytes of memory to store a two-hour movie. Now consider a newer HD TV with a screen resolution of 1920 x 1080 pixels and you can see that the amount of memory required to store and transmit video is significant!

There are methods to reduce or compress the number of bytes needed. For example, one might only store the differences from frame to frame, or, if large areas are all one color, one could store the boundaries of the area and the color, rather than the color for each pixel in the area. Compression works well for pictures of “talking heads”. It does not work well when the camera is panning a scene.

## 6.9 *Representing Sound*

To store sound digitally, we use numbers to represent the height or amplitude of the sound wave. The sound wave is measured at certain intervals and these measurements are stored, and later used to reconstruct the wave.



This wave might be stored as 6.5, 6.8, 7.3, 7.2, 7.2, 6.9, 5.5, -2.3, -3.1, etc. Note that this only approximates the original wave. The human ear can detect 20,000 cycles per second. A normal CD takes approximately 44,000 samples per second, so the sound is extremely close to what we would be able to detect however some overtones may be lost. To get the sound back, the computer sends a digital signal to a filter 44,000 times per second. The signal gives the amplitude of the sound wave. The filter smooths out the curve and then sends the necessary voltages to a speaker to reproduce the sound.

As with pictures, sound takes up a great deal of storage space and thus necessitates the use of compression. One popular compression technique is MP3 which can produce an audio file one tenth the size of the original. This has allowed for musical recordings small enough to be transferred over the internet, changing the patterns of distribution of music dramatically.

There are several advantages to the digital storage of both pictures and sound. Once either is converted to digital storage, it can easily be modified by simply changing the bit pattern. Thus coughs or pops can be edited out of music and flaws can be removed from pictures. One can also superimpose one sound on another, or one picture on another, resulting in new sounds or pictures. A drawback of this possibility is that, while it can be used to produce wonderful fantasy images, such as Forrest Gump shaking the hand of President Kennedy, it also means that pictures can no longer be accepted as necessarily accurate documentation of an event.

## 6.10 Conclusion

We have seen ways to encode integer and real numbers, text and characters, sound waves, and pictures. Each of these take various forms of data and encode them as binary strings of 0s and 1s. These 0s and 1s can be represented by high and low voltages in the bits of a computer's registers.

Note that a pattern in a register could represent a variety of things. The binary pattern 01000001 could represent the two's complement number 65, the ASCII code for A, the floating point number 17/16, part of a picture, or part of a sound wave. As we will see in the next chapter, it could also be an instruction for the computer. Naturally there must be some way of distinguishing among these various interpretations for a given binary pattern. This distinguishing is done by the programs that are written to process data. Telling the computer what kind of data it is reading is an important part of all computer programs and applications. For now, it is important to realize that data are not independent of the programs that process them.



# Chapter 7

## The CPU and Machine Language

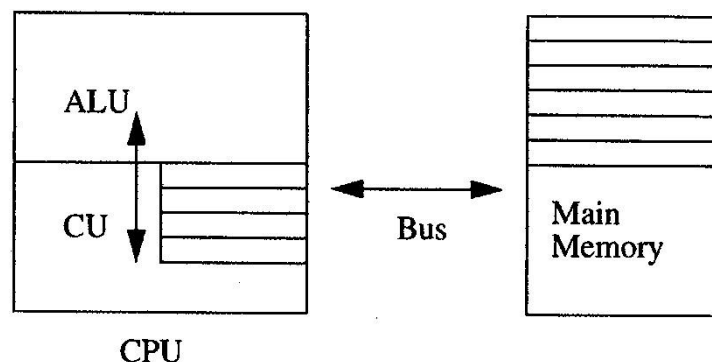
We have seen how to program a computer in a higher level language and have used a couple of the more common business applications, but how does the computer make sense out of that high level code and “run” it? This chapter offers a look into what is going on “inside” the machine when it is running programs and what has to happen before your high level program can be run by the computer.

In the very first computers, designed in the early 1940s, one actually had to run wires from the registers that held a problem's input, to the circuits being used to compute that input. When a different problem was to be solved the computer would be completely rewired. Mathematician John von Neumann published a report in 1945 that brought together ideas on how to encode and store programs in memory along with the data for those programs. This concept of storing programs right along with data became the basis for what is now known as the von *Neumann design* for computers. The beauty of this design is that anything that can be written in a program can be stored and processed by a computer without changing the computer's hardware. This allows for the tremendous versatility of the computer. While von Neumann design is not the only way to build a computer (an example of another recently developed design is the neural net), von Neumann design is the one most commonly used for general purpose computers.

### 7.1 The CPU

The circuitry that performs operations on data is located in the central processing unit, or CPU. In most computers today the CPU is housed on a single chip called a *microprocessor*. The ability to put the whole processor on a single chip accounts for both the speed and the low cost of today's personal computers and workstations. The basic design of the CPU, however, is independent of whether it is located on a single chip or is composed of several integrated circuits.

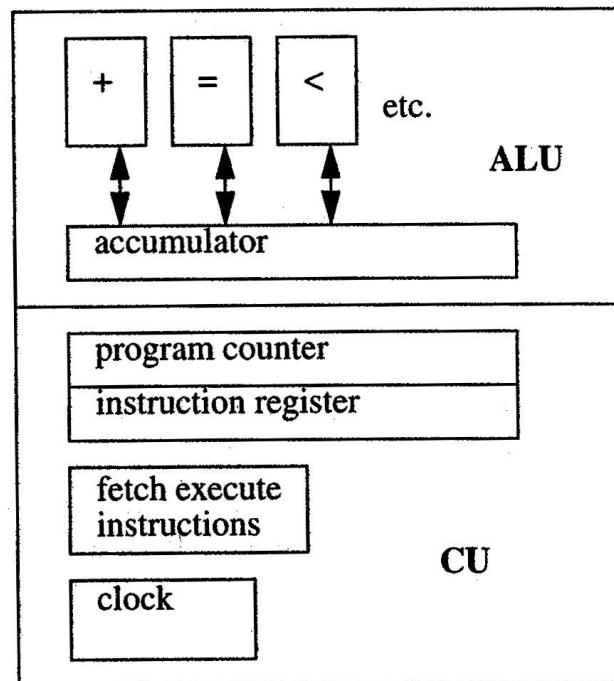
Most CPUs have two basic parts:



The CPU is connected to the main memory by a bundle of wires called a *bus*. The bus copies bit patterns between the registers in main memory and the registers in the CPU. The width of the bus determines how many characters it can transfer at once. For example, a 32-bit bus will allow 32 bits of data to be transferred at the same time. Thus the higher the bus width the faster the transfer rate between memory and the CPU.

The *control unit*, or CU, is the central part of the CPU. It oversees the transfer of information between all the components of main memory and the CPU. The control unit also accesses the information in ROM to boot the computer and get it ready for processing.

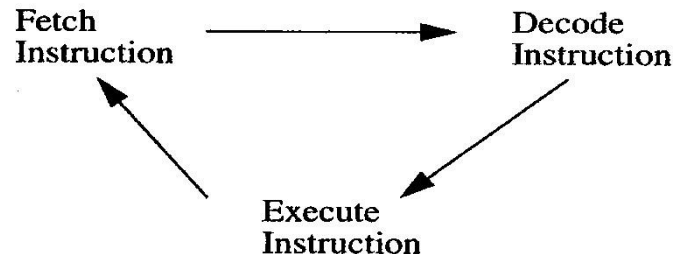
The *arithmetic-logic unit*, or ALU, contains the circuits to do the actual computations. Circuits are built into the ALU for each elementary operation of the computer, including arithmetic operations such as addition, subtraction, multiplication and division, and logical operations such as equivalence, less than, greater than, and negation. A special register in the ALU, the *accumulator*, which we will sometimes abbreviate as AC, holds the results of these calculations.



The CPU processes information as follows. Suppose we wish to add two numbers stored in RAM. The control unit would receive the instruction to add and the location in RAM of the two numbers to be added. The bit pattern representing the first numbers would be transferred via the bus to the accumulator. The bit pattern of the second number would then be added to the pattern in the accumulator, starting at the rightmost bit, by the addition circuit in the ALU. The output from the circuit would go into the accumulator and then be transferred via the bus to a register in main memory.

## 7.2 The Fetch-Execute Cycle

A more complex program would have a sequence of several operations or instructions. The control unit oversees the *fetch-execute cycle*, in which it fetches one instruction from main memory, decodes the instruction, oversees the execution of that instruction, then fetches the next instruction, continuing the process until it receives an instruction to stop.



In order to do this the CU has two special registers. The *program counter* contains the address of the next instruction to fetch. At the beginning of a program this would be the location in memory of the first instruction in the program. After an instruction has been fetched the address in the program counter is automatically increased by one, unless the instruction fetched tells the control unit to do otherwise. The *instruction register* holds the current instruction.

A clock controls the timing of the fetch-execute cycle. Each computer contains a clock chip which generates a steady stream of regular electrical pulses. At each clock pulse, the processor proceeds by one step. A step might be fetching a new instruction, or sending two inputs to the ALU to be added. The more frequent the clock pulses, the faster the processor. Processors on the market typically run with clock speeds of 3 to 4 GHz, (gigahertz or one billion pulses per second).

## 7.3 Machine Language

The instruction set programs can be written in is dependent on the make-up of the processor. In other words, each machine has a set of operations built into the ALU. These basic operations are each encoded with a binary bit pattern, just as we saw data could be encoded in chapter 6. The CU has instructions that allow it to decode each of these patterns and send the proper input to the required circuit in the ALU. The set of operations, encoded in binary, for a given machine is called the *machine language* for that computer. Each processor has its own machine language.

### RISC and CISC Architectures

Different processors on the market today have different sets of operations built into them, and hence, different machine languages. Two basic approaches to processor design are to build a large number of instructions into the ALU, which allows for rapid processing on a variety of instructions, but a slower control unit, since the control unit has more operations it must distinguish among, or to have a small set of basic instructions, building more complex instructions by using several of the basic instructions.

A CPU with a CISC (Complex Instruction Set Computer) architecture has a large instruction set and therefore a large number of circuits in the ALU whereas a RISC (Reduced Instruction Set Computer) architecture has fewer instructions to choose from and is therefore faster than CISC machines because it takes less time for the control unit to decode an instruction when there are fewer to choose from and, although some instructions that are missing in a RISC machine take more time to execute, it turns out that 20% of the instructions are used 80% of the time. So the missing instructions are needed only rarely. Because of this speed advantage, most processors today are moving toward using some features of both approaches to get the best performance possible.

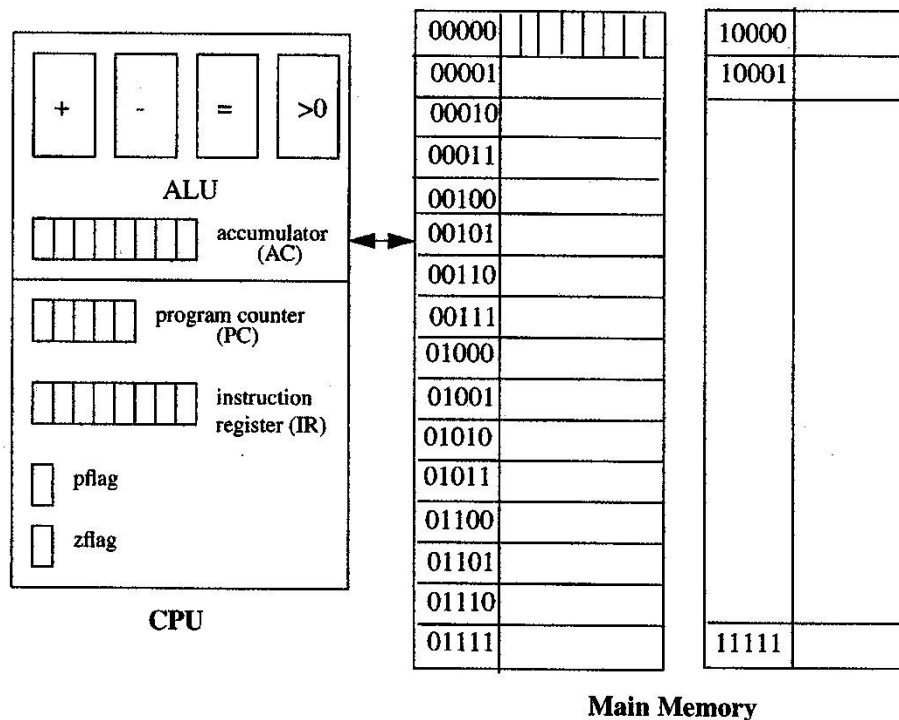
## Machine Instructions

The instructions found in a typical machine language fall into three categories: data transfer, arithmetic/logic operations, and instruction control. *Data transfer commands* copy data from one location to another. For example, a LOAD command would copy data from a given memory register into the accumulator while a STORE command would copy the pattern currently in the accumulator into a designated memory register. *Arithmetic/logic commands* call upon the circuits in the arithmetic/logic unit to transform data. Examples include ADD and SUBTRACT. *Control commands* alter the contents of the program counter. Instead of proceeding to the next instruction in memory, instruction control commands designate which instruction to go to next. These instructions are also known as “jump” or “branch” instructions and they can appear in two forms. Conditional jumps occur only when the contents of the accumulator meets a certain condition, such as when the accumulator holds a zero, or when the value in the accumulator is positive. Unconditional jumps occur no matter what is in the accumulator.

## 7.4 A Simple 8-bit Computer

We will look at the design of a simple 8-bit computer in order to see how machine language programs work. This computer has many registers to hold data and/or instructions. Our machine has 32 8-bit registers in main memory, with 5-bit addresses ranging from 00000 (0) to 11111 (31). In the CPU we have a 5-bit program counter to hold the address of the next instruction to be fetched. The program counter begins at 0. There is also an 8-bit instruction register holding the instruction currently being decoded and executed, an 8-bit register called the accumulator that will hold the current data loaded from memory or the results of arithmetic operations, and two 1-bit flag registers. The zflag register shows whether the contents of the accumulator is zero by containing a 1 when it is and a 0 when it is not, and the pflag register contains a 1 when the contents of the accumulator is a positive number and a zero otherwise.

Our computer can carry out four operations in the ALU: addition, subtraction, determining if the contents of the accumulator is 0, and determining if the contents of the accumulator is positive. Because we are limited to eight bits and to keep the design simple, our machine will only accept positive and negative integers in 2's complement encoding as data.

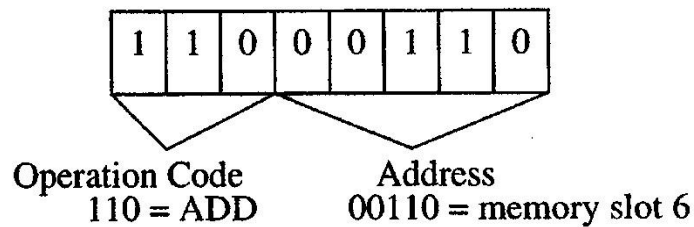


The machine language for this computer consists of eight commands or operations. Since we only need three bits to have eight distinct binary codes (000 up to 111) this will leave 5 bits for the address of the data the instruction is to work on. In other words, the first three bits of each instruction will tell what operation to do and the last five bits are available for the address of that data to be operated on. When a command does not need data, the last five bits will be ignored.

### Machine Language Command Set

<i>Operation Code</i>	<i>Meaning</i>	<i>Description</i>
000	HALT	stop execution of program
001	JUMP	place address given in PC
010	JZER	place address given into PC only if AC=0
011	JPOS	place address given into PC only if AC>0
100	LOAD	copy contents of address given into AC
101	STORE	copy contents of AC into address given
110	ADD	add contents of address given to AC
111	SUB	subtract contents of address given from AC

Here is a sample machine language instruction to add the contents of memory slot 6 to whatever is in the accumulator.



This machine has four control commands, HALT, JUMP, JZER, and JPOS. HALT is the only command that does not require an address in main memory in the rightmost five bits. The CU will stop execution when it sees 000 regardless of what is in the remaining bits. Thus 00000000, 00011111, and 00010100 all have the same meaning. The three jump commands alter the normal flow of the program. Recall that once an instruction has been fetched and executed the program counter automatically is increased by one, thus moving the computer on to the next instruction in the memory. We may not always want to do the next instruction. Sometimes we may wish to skip an instruction or go back and repeat a previous one. The jump commands allow us to do this. JUMP makes the next instruction the one in the memory address given. Thus 00100011 would mean that the next instruction that will be executed is the instruction in memory address 00011 (3). JPOS and JZER add conditions to when to jump. When a JPOS command is encountered the control unit checks the pflag. If there is a 1 in this flag it means the contents of the accumulator is positive so the command at the address given becomes the next command. If the pflag is a 0 then the contents of the accumulator is either zero or negative and so the program counter is automatically increased by one and this command is effectively ignored. JZER works the same way, checking the zflag to see if the contents of the accumulator is zero and jumping when it is.

The two data transfer commands are LOAD and STORE. LOAD copies the contents of the address given into the accumulator. Note that this does not alter the contents in the memory address. 10001000 would copy whatever the contents of memory register 01000 are into the accumulator, leaving the contents of 01000 unchanged. STORE reverses this process, replacing whatever was in that address previously with the contents of the accumulator, leaving the contents of the accumulator unchanged.

We also have two arithmetic/logic commands, ADD and SUB. ADD adds the contents of the address given to whatever is already in the accumulator. SUB subtracts the contents of the address given from whatever is in the accumulator. Thus to add two numbers we must first LOAD the first number into the accumulator and then ADD the second number.

## Input and Output

Our simple computer has 32 memory registers for holding the programs and data, however, two of these memory cells have a special purpose in order to have input and output on our machine. The two cells, 30 and 31, exist within the machine but are not visible in the simulator. Cell 11110 (30) is a designated input cell. A LOAD command with this address will cause the machine to access an input device. In practice, the simulator will pause and wait for the user to type in a value and press Enter before it proceeds to the next instruction. (The value is displayed in the I/O window and stored in the hidden cell 30.) Similarly, cell 11111 (31) is a designated output cell. A STORE command with this address will copy the contents of the accumulator to this address,

which in our case is mapped to the output window where the value will be displayed. The use of these two special cells in this way is called memory-mapped I/O.

### Machine Language Programs

Programs for this simple computer consist of a series of commands ending in a HALT command. Each program will be loaded into main memory beginning with address 0. Commands and data for the program are stored in consecutive memory addresses.

Let's begin with a program to compute  $3 + 4 + 5$  and store the sum in memory at address 8, where the three numbers to be added have already been placed in memory at addresses 5, 6, and 7. We will need to load the 3 into the accumulator, add the 4, add the 5, and store the answer. Our algorithm for this program would be:

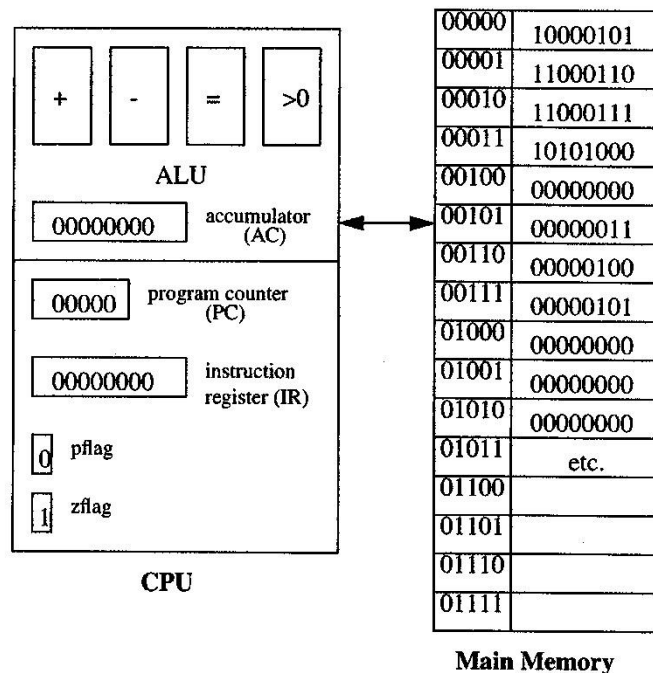
1. add 3 + 4
2. add 5 to that sum
3. store the answer
4. end

The machine language version of the program would be:

<i>address</i>	<i>contents</i>	<i>meaning</i>	
00000	10000101	LOAD 5	(load the contents of address 5 into the AC)
00001	11000110	ADD 6	(add the contents of address 6 to the AC)
00010	11000111	ADD 7	(add the contents of address 7 to the AC)
00011	10101000	STORE 8	(store the contents of AC in address 8)
00100	00000000	HALT	
00101	00000011	3	
00110	00000100	4	
00111	00000101	5	
01000	00000000		

Since the program will be loaded into memory starting with slot 0, the first piece of data, the 3 will be in cell 5, the 4 in cell 6, and the 5 in cell 7. We will store the answer in cell 8. At the start of this program the computer would look as follows:





The 0 in the program counter means that the computer will begin execution by fetching the command in memory cell 0. This command will be transferred into the instruction register, decoded, and executed, resulting in the copying of the 3 in memory cell 5 into the accumulator. Then the program counter is automatically incremented by 1. The computer now fetches the command in memory cell 1. This fetch-execute cycle continues, adding the 4 and the 5 to the 3 and storing the result in cell 8. When the computer decodes the HALT command from cell 4 execution of the program ends.

## 7.5 Assembly Language

Formally, machine language code consists of the binary instructions which can be directly executed by the processor. As you might imagine, long programs in machine code are quite difficult to read. To make them easier to read and understand, assembly language was created. Assembly language has a one-to-one correspondence to machine code. It uses a one-word command for each three-bit machine code command, and decimal numbers for addresses and data. However, assembly language code must be translated into machine language code in order to be executable by the processor. The program that translates assembly language to machine language is called an *assembler*. The assembler has two basic parts to it. First it takes the assembly language version of the program, called the *source code*, and looks up each word, substituting the corresponding machine code as it creates a new machine language version of the assembly code. It also converts all the decimal numbers into two's complement. The final machine code version of the program is called the *object code*.

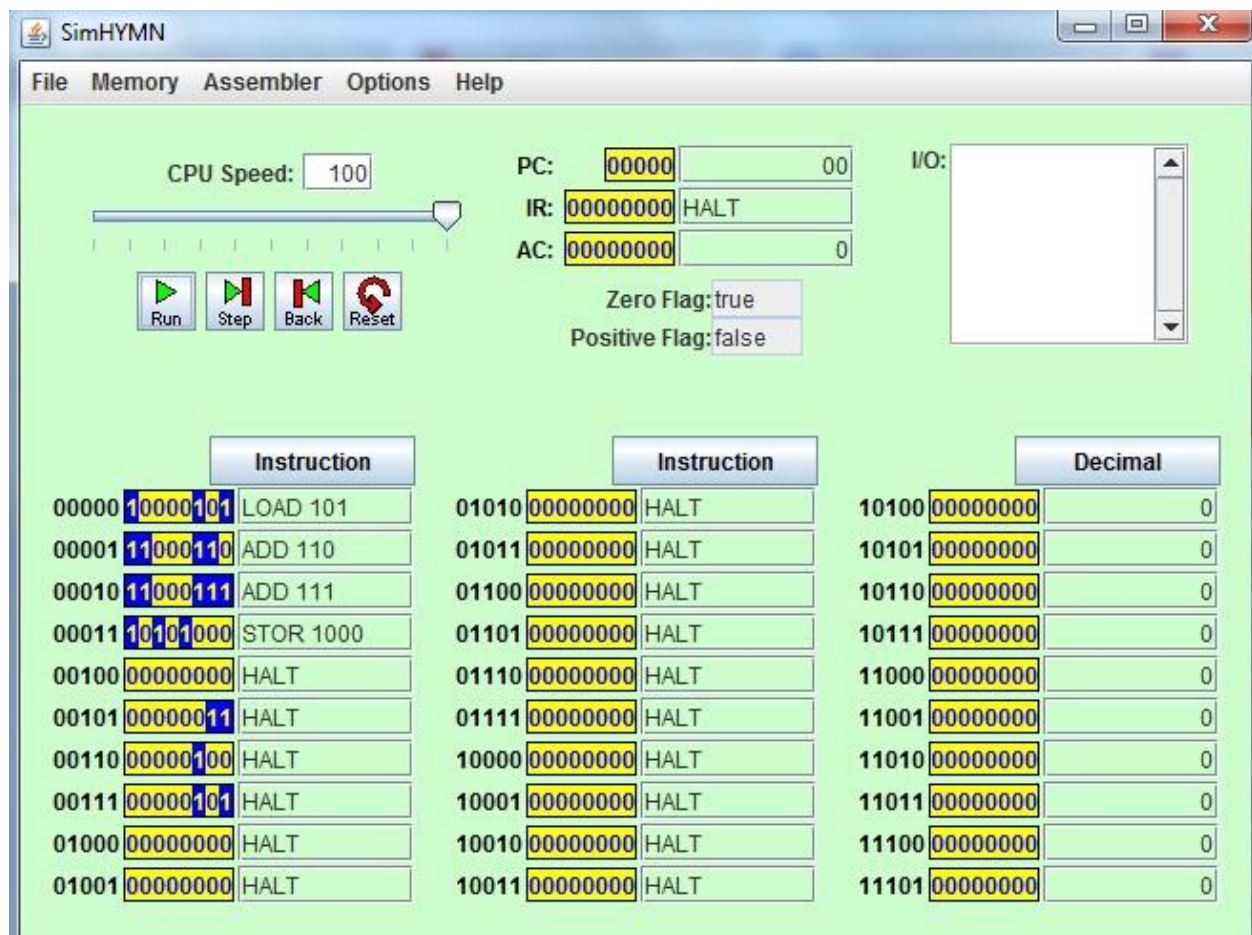
There is one other difficulty the assembler must deal with. We have been writing all our programs with the assumption that they would be loaded into the memory starting in cell 0. In a large computer system with multiple users this isn't always possible. Users' programs may end

up anywhere in memory, and in different locations each time they are run. Thus most assemblers produce what is known as relocatable object code. This code substitutes variables for the cell addresses (i.e. FirstNumber instead of 11). The assembler then passes the object code on to a program called a loader. The loader finds a suitable space in memory to load the program and substitutes the appropriate cell addresses for the variables in the program. It then copies the address of the first instruction into the program counter and the program is ready to run.

The machine language program from the previous section has been rewritten in Assembly Language, and is shown in the next image, this time using labels to refer to address locations. Labels are used in assembly language as a way to refer to a location to jump to and also as a way to refer to locations in memory where data has been stored. When referring to data, the labels are essentially used as variable names. The use of labels in the assembly code make it easier to program without needing to know the binary address and the assembler does the work of replacing those labels with the address they refer to when the code is translated to binary. Also note that the locations to be used as variables need to be declared at the bottom of your assembly code and given an initial value. Sometimes that value will change when your program is run and other times it remains a constant, depending on your algorithm. Also note that every label used is followed by a colon, :, to indicate to the assembler program that that location in memory has a name associated with it. (ie. x: 0)

```
# This program will calculate the sum of three
# numbers listed in the program and store the sum in memory.
LOAD  FIRSTNUM
ADD   SECONDNUM
ADD   THIRNUM
STORE SUM
HALT
FIRSTNUM: 3
SECONDNUM: 4
THIRNUM: 5
SUM:      0
```

Next is a screen shot of the SimHYMN simulator showing the machine language version of the program after an assembler program has translated it into machine language. Note how the assembler has translated the labels (variable names) to the corresponding address of the locations in memory. The assembly instruction, ADD SECONDNUM, was translated to ADD 110. 110 is address 6 in memory where the second number is stored.



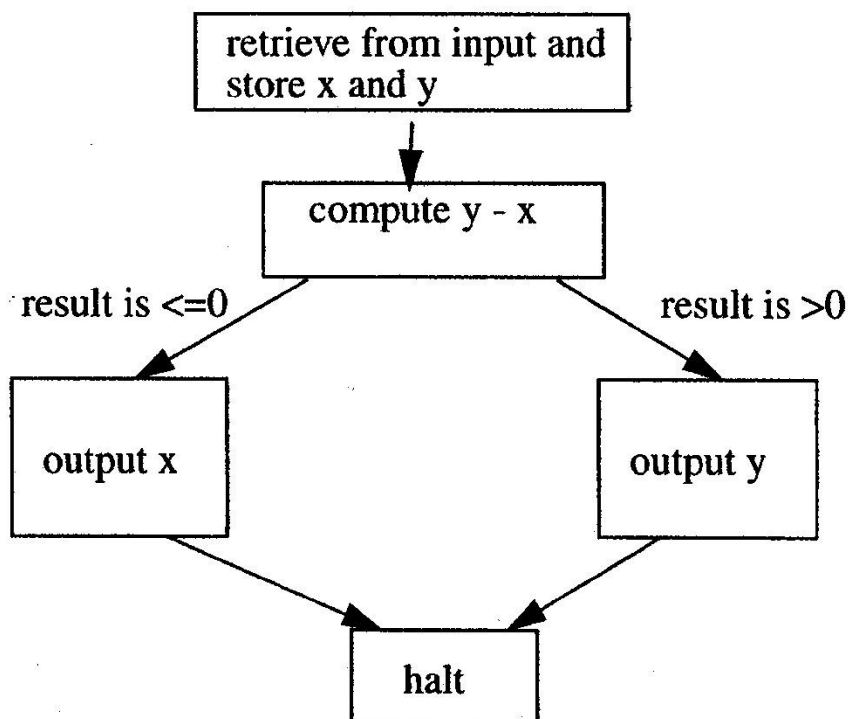
Next is the Visual Basic (VBA) Version of the above program:

```
Sub addThree()
    'This program will calculate and store the sum of three
    'numbers given in the program.
    Dim FIRSTNUM As Integer, SECONDNUM As Integer
    Dim THIRDNUM As Integer, SUM As Integer
    FIRSTNUM = 3
    SECONDNUM = 4
    THIRDNUM = 5
    SUM = 0
    SUM = FIRSTNUM + SECONDNUM
    SUM = SUM + THIRDNUM
    'MsgBox (SUM) 'without this line you can't see the result
End Sub
```

Let's look at another example. Suppose we wished to find the larger of two inputted numbers, x and y. We can do this by subtracting one from the other and seeing if the result is positive or negative. Since the numbers are coming from input, both will be loaded in from the same cell, cell 30. Both also need to be copied to other locations in the memory so that we have them for output later.

0	LOAD 30	loads first number
1	STORE 11	first number is now in cell 11
2	LOAD 30	loads second number from input
3	STORE 12	second number is now in cell 12
4	SUB 11	subtracts the first number from the second
5	JPOS 8	if the result is positive, jump to address 8 for next instruction
6	LOAD 11	result was not positive, first number largest, put in accumulator
7	JUMP 9	jump to instruction that will display the result
8	LOAD 12	result was positive, second number largest, put in accumulator
9	STORE 31	biggest number sent to output
10	HALT	

This program contains a *branch*, in that sometimes we wish to execute lines 6, 7, and 9, and sometimes we wish to execute lines 8 and 9, but never all 4. If we were to look at the structure of this program graphically we would see two possible paths. A visual model of a programs structure, such as the following is called a *flowchart*.



As you have seen in a previous assembly language program, the labels are essentially used as variable names when referring to data. There are also some words used in assembly languages that have special meanings, similar to reserved words in VB. Two of those words in this assembly language are READ and WRITE. Note the use of READ and WRITE in place of Load 30 and Store 31 in the assembly language programs that follow. READ is a single word command to get input from the user via the keyboard and WRITE is a single word command to display the contents of the accumulator in the output window. The following program is a new version of the previous program to determine the biggest of two numbers input by the user, this time making use of labels/variables for data and the special command labels, READ and WRITE.

```

# This Assembly Language Program gets two numbers from the user,
# determines which one is bigger, and displays the value of the bigger number.
READ                                #get first number
STOR      x                        # store first number
READ                                #get second number
STOR      y                        # store second number
SUB       x                        # subtract first number from second number
JPOS      yIsBigger                # if difference is positive, go to display second number
LOAD      x                        # if difference is 0 or less, load first number to display it
JUMP End
yIsBigger:  LOAD      y            # load second number
End:        WRITE     # display contents of the accumulator
            HALT      # end program

x:    0
y:    0

```

## Loops and Branches

Now let's consider a program to add all the numbers between 1 and 10 and to display their sum. We could simply LOAD a 1 and then have 9 add statements followed by printing the sum. The algorithm for this method of addition would be:

1. begin with 1
2. add 2
3. add 3
4. add 4
- etc.
10. add 10
11. display the sum
12. end

But this would hardly be practical, especially if we were adding the numbers between 1 and 100, or 1 and 1000. A better way is to use a jump command to repeat the adding process the requisite number of times. We would use an algorithm something like this:

1. begin with 0 as SUM, 1 as X
2. add X to SUM
3. add 1 to X
4. go back to step 2
5. display the sum
6. end

The assembly language version of this would be the following:

```

0    LOAD  SUM
1    ADD   X                adds a number into the sum
2    STORE SUM
3    LOAD  X
4    ADD   ONE              adds 1 to get the next number
5    STORE X
6    JUMP  0                returns to beginning
7    HALT

```

8	SUM:	0
9	X:	1
10	ONE:	1

This program is keeping the sum of the numbers as we add them in SUM, while X counts up by 1 each time through the program. Thus the first time through it would add the 1 in X to the 0 in SUM, store the resulting 1 in SUM, then add the 1 in ONE to the 1 in X, storing a 2 in X. The jump command would take it back to the LOAD in cell 0 and the whole program would repeat, this time adding the 2 in X to the 1 in SUM, storing the resulting 3 in SUM, then adding 1 to the 2 in X. As you can see X in cell 9 is holding the numbers 1, 2, 3, 4, . . . while SUM in cell 8 is holding the sums 0, 1, 3, 6, . . . The problem with this program is that it will never halt. Every time it gets to the jump command it will go back to cell 0. Sections of code that repeat are known as *loops*. Code that repeats forever is known as an *infinite loop*.

This shows that often our first attempt at an algorithm needs a bit of refinement. We need a way of making sure we stop when we get to 10. We could store a 10 and compare it to the number in X, stopping when we get to 10, but since we have a JPOS command that jumps only when we have a positive number in the accumulator, let's count down from 10 to 1 instead of counting up from 1 to 10. This way when we reach a 0 in X we will halt. So we want to start with a 10 in X and subtract a 1 each time through the loop.

Now our algorithm would be:

1. begin with SUM = 0 and X = 10
2. add X to SUM
3. subtract 1 from X
4. if X > 0 then go back to step 2
5. display the sum
6. end

The assembly language version of this would be:

0	LOAD	SUM	
1	ADD	X	# adds a number to the sum
2	STORE	SUM	
3	LOAD	X	
4	SUB	ONE	# subtracts 1 from X to get next number
5	STORE	X	# store new value of X
6	JPOS	0	# returns to beginning while X > 0
7	LOAD	SUM	# put contents of SUM in the accumulator
8	WRITE		# display the sum
7	HALT		
8	SUM:	0	
9	X:	10	
10	ONE:	1	

Notice that this program will add the numbers  $10 + 9 + 8 + \dots + 1$ . When X in cell 9 has a 1 in it, the 1 is added to SUM, then 1 is subtracted in line 4, leaving 0 in the accumulator. The JPOS will not jump when the accumulator holds a 0 or a negative number, so the program moves on to

the LOAD SUM statement. The final sum is in SUM in cell 8. To output this sum we need to first load it into the accumulator and then we can display it using WRITE.

Suppose we wished to add only the even numbers between 10 and 1. Note that the same program would do the job for us. All we would need to change is the contents of cell 10 to 2. (You would probably also want to change the label ONE to TWO to make your code easier to understand.)

Next is another sample assembly program that uses a loop to compute the sum of the numbers from N down to 1, where the value of N is input by the user as part of the program. This program also uses an additional label in a JPOS instruction to tell the computer where to jump to. Remember that the assembler replaces all labels with the appropriate cell address in memory.

# This Assembly Language Program will calculate and display the sum  
# of the numbers from N down to N, where N is input by the user.  
# The assumption is made that N is a positive integer.

```

READ                                # get a positive integer from the user
STORE      N                       # store the value in accumulator in N
addAgain:  ADD      SUM             # add the value of N to SUM, (SUM is intially 0)
           STORE    SUM            # save new value of SUM
           LOAD     N              # decrement N
           SUB      ONE
           STORE    N              # store new lower value of N
           JPOS     addAgain        # repeat addition while N > 0
LOAD SUM                             # get value of SUM from memory and put in AC
WRITE                                           # display contents of accumulator (your result)
HALT                                           # end program
N:      0                                     #initial values of variables
SUM: 0
ONE: 1

```

Here is the VB version of the previous program:

'This program gets an integer, N, from the user  
'and then calculates and displays the sum of the integers  
'from N down to 1

```

Dim Sum As Integer = 0, N As Integer = 0, One As Integer = 1
N = InputBox("Enter an integer", "N")
Do
    Sum = Sum + N
    N = N - One
Loop While N > 0
MsgBox("The sum is " & Sum)

```

On the next page is a sample algorithm and the associated VB and assembly language programs. Note again, the use of READ and WRITE in place of Load 30 and Store 31 in the assembly language program.



```

# Here is an algorithm and a program that will accept six numbers from the user
# and count how many of the numbers are even. The program will display that count.
# algorithm:
#   1. initialize evenCtr to 0
#   2. initialize listCtr to 6
#   3. get a number from the user, call it X
#   4. if X is even, add 1 to evenCtr
#   5. decrement listCtr by 1
#   6. if listCtr is positive, then go back to step 3
#   7. display evenCtr
#   8. Halt
# VB program -----
# Private Sub cmdCountEvens_Click()
#   'this program gets six integers from the user and counts how many are even.
#   'declare variables
#   Dim listCtr As Integer, evenCtr As Integer, X As Integer
#   evenCtr = 0
#   listCtr = 6
#
#   Do 'start loop that will get six numbers from user
#       X = InputBox("ENTER A NUMBER") 'get a number from the user
#       Do 'check to see if it is even
#           X = X - 2 'repeat subtraction
#       Loop While X > 0 'check to see if it is even
#       If X = 0 Then 'if zero, x was even
#           evenCtr = evenCtr + 1 ' if even, then increment evenCtr
#       End If
#       listCtr = listCtr -1
#   Loop While listCtr > 0
#   'display results
#   MsgBox "There were " & evenCtr & " even numbers in your list."
# End Sub

# assembly language program -----
GET_A_NUMBER:  READ      # get a # from the user, (X)
SUBTRACT_TWO:  SUB TWO # subtract two repeatedly to determine if X is even
               JPOS SUBTRACT_TWO      # if acc is positive, subtract again
               JZER INCR_evenCtr      # if accumulator is 0, then X was even
DECR_listCtr:  LOAD listCtr          # decrement listCtr
               SUB ONE
               STORE listCtr
               JPOS GET_A_NUMBER      #if acc is positive, get another number
               LOAD evenCtr           # load and then display evenCtr
               WRITE
               HALT
INCR_evenCtr:  LOAD evenCtr          # add one to the evenCtr
               ADD ONE
               STORE evenCtr
               JUMP DECR_listCtr      # go back to DECR_listCtr

TWO:          2
ONE:           1
listCtr:       6
evenCtr:       0

```

## **7.6 Conclusion**

In a von Neumann computer programs are coded and stored in memory just like data. Each computer has a specific code, called machine code, which contains binary bit patterns that corresponds to the operations available in its ALU. The computer fetches one instruction at a time, decodes it, executes it, and then fetches the next. Because machine code is difficult for humans to read, programmers generally use assembly language in which words are substituted for the machine code patterns.

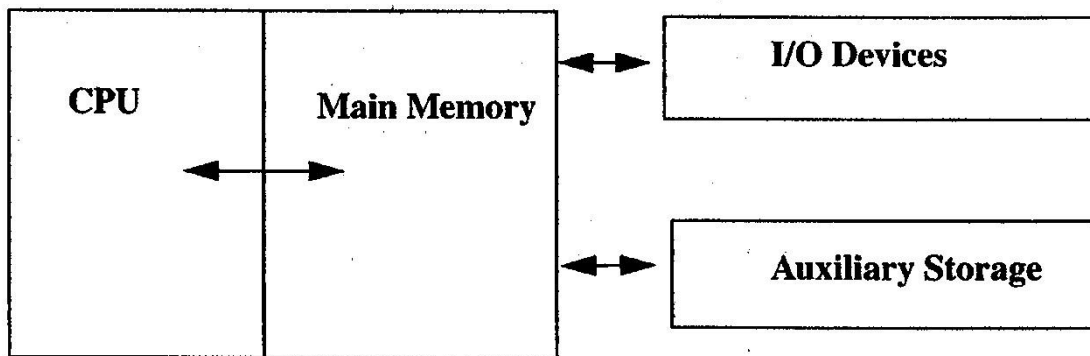
Storing programs as data has several advantages. It allows programs and data to reside in the same memory locations, rather than needing separate areas. It also allows programs to be treated as data, meaning that one program could modify another or even itself.

## Chapter 8

# Computer Hardware, Operating Systems and Networks

### 8.1 Overview of Computer Hardware

Recall that the hardware of the computer is made up of four categories of devices:



The CPU and main memory are a part of every computer. In personal computers the CPU is a single silicon chip with circuits etched on it known as the *microprocessor*. It is mounted on a circuit board known as the *motherboard*.

The motherboard is connected to the main memory, which is divided into at least two different sections. Random Access Memory, or *RAM* consists of registers that store information before or after processing by the CPU. This is the part of memory available to the user and its contents are short term and constantly changing. RAM memory is said to be *volatile*, which means it is dependent on a continual supply of power to its circuits, therefore if the computer is shut off or the power goes out everything in RAM memory is lost. This is why you should frequently save your programs and data on longer term storage media such as flash media or disks. We also have some things that we would like to have permanently stored in memory, thus there is a second section of main memory called Read Only Memory or *ROM*. This part of memory is not volatile and holds the programs that are necessary for the basic operation of the computer system, such as the boot program, which starts up the computer when it is first turned on. Information in this part of the memory is permanently etched on a chip; it can be read but not altered.

Another type of memory you may occasionally hear mentioned is *cache* memory. Cache memory functions just like RAM, however, it is on a smaller and faster (and more expensive) chip. Cache memory is often used to hold data and instructions temporarily on their way into or

out of the CPU. Flash memory is like cache, but nonvolatile. It is faster than RAM and used in devices such as mobile phones.

## **8.2    *Input and Output***

### **Input**

The RAM portion of main memory receives data from input and output devices. Input devices accept data from the user and convert it into binary code before sending it to the main memory. Common input devices include the touch screens, keyboards, mouse, scanners and digital cameras, microphones, and various sensors. What type of input devices you might want for a given computer depends entirely on the tasks that computer will be used for. Most general purpose computers come with at least a keyboard and a mouse.

### **Output**

Output devices include monitors, speakers, and printers. There are two main types of monitors, CRTs (cathode ray tubes), which in the past were the standard monitors for desktops, and Flatpanel displays, which are now the standard display for both laptops and desktops. Flat-panel displays were not widely used when they were first developed because of their price, which was 4-8 times as much as that of a CRT, but the price has dropped to make them more affordable. When buying a monitor there are three things to take into account. The resolution is the number of pixels on the screen. The more pixels per square inch the clearer the image. Standard screen resolutions range from 640 x 480 to 1600 x 1200. The dot pitch of a monitor determines how close or far apart the pixels are. The closer the pixels, the clearer the image. Standard dot pitches tend to range between .2 and .3 millimeters. The refresh rate determines the number of times per second that a new electric pulse is sent to each pixel, keeping its color from fading. A typical refresh rate is 70Hz or higher.

The most common printers today include the laser printer and the ink-jet printer. Both form images of characters and graphics through the use of small dots. The difference is that an ink-jet printer sprays small dots of ink onto the page from a matrix of ink jets while a laser printer creates an image of the page on a drum covered with a magnetically charged toner and then transfers the image to the page. While ink-jet printers are considerably cheaper than laser printers, they are slower in that they print the image one line at a time as opposed to printing the entire page at once.

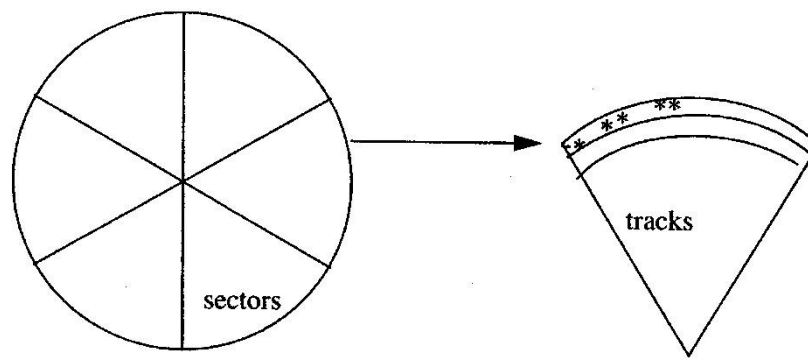
## **8.3    *Auxiliary Storage***

The purpose of auxiliary storage is to have long term storage for programs and data. Remember that RAM is volatile, thus inappropriate for long term storage. RAM is also smaller than outside storage capabilities; a computer that did not have another type of storage would be very limited in the type and number of programs it could handle. There are several common forms of

auxiliary storage in use today: magnetic disks, optical disks, flash media, Solid State Drives, (SSD), and magnetic tape.

## Magnetic Disks

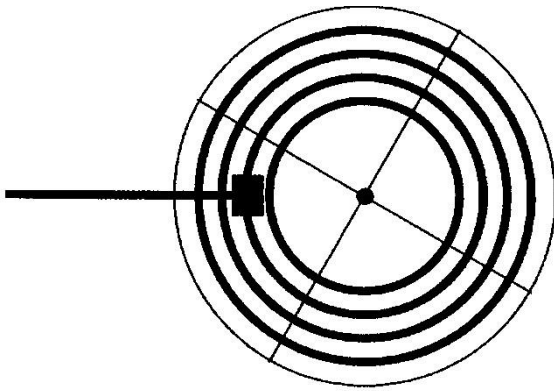
*Magnetic disks* are the most common storage medium. Like magnetic tape, disks are covered with a magnetic coating that takes a charge in order to hold information. The information is stored on concentric circles on the disk called *cylinders* or *tracks*. A track contains more information than we would want to read into RAM at one time, so each cylinder is divided into pie-shaped areas called *sectors*. Data is stored along a track in a single sector as in the figure below:



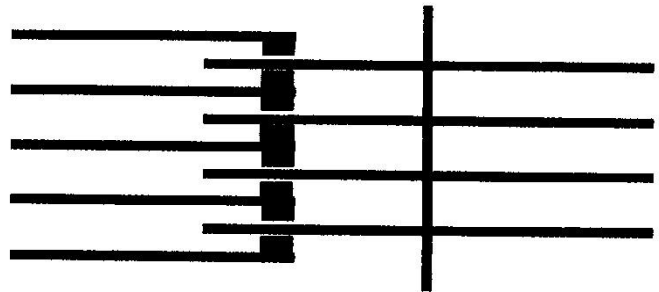
Each track contains the same number of bits per sector. These bits are stored more compactly toward the center of the disk. A read/write head can be moved to any position on the disk to locate desired information. This makes disks the medium of choice for storing files that are currently in use. Because the read/write head can go directly to any spot on the disk, this method of storage is called *random access*.

Magnetic disks may be portable or fixed. A fixed or “hard” disk consists of a metal or glass disk covered with a magnetic coating and housed in a sealed unit. Hard disks may be single disks, as on a personal computer, or they may consist of up to ten or twelve disks mounted on a single spindle for storage on larger computer systems. Hard disk units have the capacity to store thousands of gigabytes of data, (1024 gigabytes = 1 terabyte), on a single disk and the capacity continues to grow as suggested by “Moore’s Law” that states that the transistor capacity on a single chip will double every two years. Storage capacity has generally followed this sort of growth pattern also since 1965 when Gordon Moore made that statement and is currently into the terabyte capacity, ( $2^{40}$  bytes) for hard drives. Before a disk can be used it must be *formatted*. Formatting sets up the pattern of sectors and tracks on a disk. It also establishes a directory on part of the disk where the computer can keep track of the sector and track number where each file is stored. Formatting destroys all the information that is on a disk at that time. Portable disks called diskettes or floppy disks (because the disk inside the case is flexible) were once a common way to store and transport information. Their limited capacity of only 1.44 megabytes (MB) of data for a standard 3 1/2” diskette now makes them obsolete. While diskettes commonly had 135

tracks per inch and 18 sectors, hard disks have thousands of tracks per inch and up to 64 sectors. The read/write heads do not touch the hard disk but float slightly above it. This allows hard disks to rotate up to 100 times faster than diskettes, which allows for quicker location and transfer of data. It also accounts for the reason hard disks are a sealed unit. Any dirt that came between the disk and the read/write head would corrupt the data stored on the disk.



Disk and read/write head: top view



Disk pack: side view  
note multiple read/write heads

## Optical Disks

Another form of disk storage is the *optical* disk which has several formats including *compact* disk (CD), digital video disks, (DVD), and Blu-ray disks, (BD ... a high definition video disk). Data is stored on an optical disk as on a magnetic disk, with the bit pattern transferred to the disk as a series of dots. However, these dots are formed on an optical disk by burning minute pits in the reflective surface of a hard plastic disk. A laser beam is then used to detect these pits and recover the pattern. Pitted areas are interpreted as 0s and smooth areas as 1s. Unlike magnetic disks, the information on an optical disk is stored on a single track that spirals around the surface of the disk. The use of pits in the surface rather than magnetism has two implications for optical disks. First, it allows for a greater density of storage than the portable magnetic diskette format; optical disks hold from 600 MB to 10 GB of data. This makes optical disks the medium of choice for storing video or audio information. Second, since the actual surface of the disk is altered, most optical disks are permanent storage and cannot be reused. Some optical disk drives do allow for re-burning by first smoothing out the surface. This is a process that can be carried out on a single disk only a few times without compromising the strength of the disk.

## Flash Media and SSD

Flash media and solid state drives are memory devices that have no moving parts, but are made only of miniature electrical components which makes them more reliable and with faster retrieval

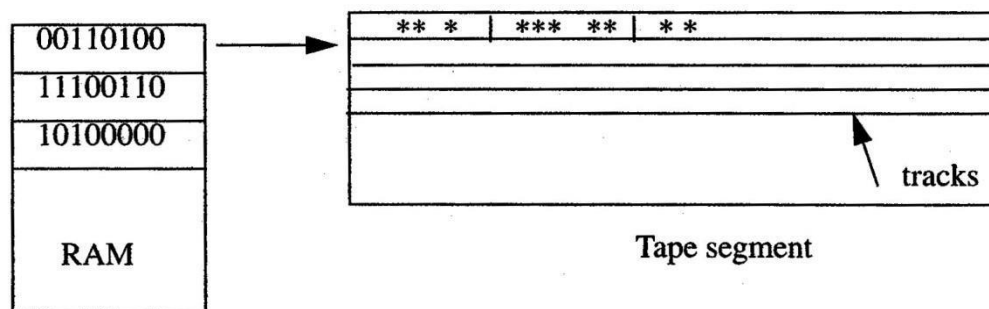
times. When this type of memory is used inside of a computer in place and in the form of the former hard drive, it is called an SSD, (solid state drive), even though it is not really a “drive”. Another common form of flash media used for portable storage are USB flash drives, also known as thumb drives, that are smaller, have a greater capacity (multiple gigabytes) than the older magnetic diskettes and have replaced the use of diskettes in the personal computer market.

Several other formats of flash media are common as the memory for cameras, cell phones, music players, etc. As capacity increases and cost decreases, solid state memory is being used more and more in many applications.

## Magnetic Tape

*Magnetic tape* was the most common auxiliary storage medium prior to the 1980s. Today it is used primarily for backup and archiving on some large computer systems. The data on the hard disks of a large system need to be downloaded periodically to a tape so that data are not lost should the disks be corrupted, which could happen through a mechanical failure or through infection with a virus. Tapes are still used for long term backup simply because they are inexpensive, easy to store, and hold massive amounts of data, but backups are now often kept on disks for quicker retrieval, since the cost of hard drives has decreased and their capacity has dramatically increased over time.

Data is stored on a magnetic tape in the same binary codes that are used to store data in RAM. A magnetic tape is a thin plastic tape covered with a coating that takes a magnetic charge. Each tape is divided into multiple segments. When a file is stored, the segment it is stored in is recorded in a directory. To access the file, the tape must be mounted on a tape drive. It is then fast forwarded to the proper segment, where information can be written to or read from the tape. There are multiple tracks on the tape that can be accessed individually. Patterns of 0s and 1s are recorded on a single track as dots of magnetic charge, a dot going in all the spots where a 1 occurs in the bit pattern in RAM.



Tapes are called *sequential access* devices because the only way to retrieve information stored on a tape is to start at the beginning of the tape and scan forward until the desired information is located. Since this can be very time consuming, tapes are rarely used for storage of current information.



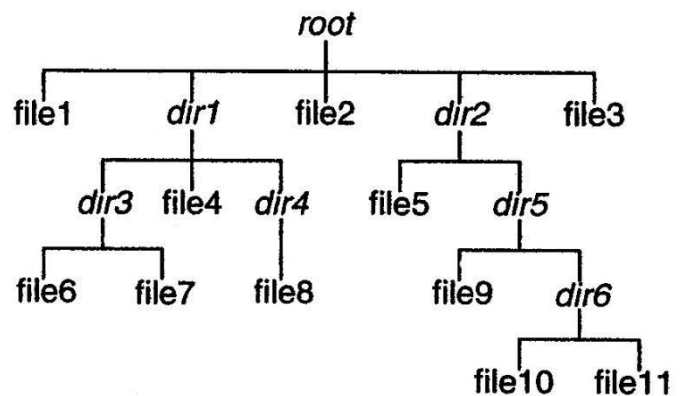
## File Management

It is easiest for the computer to always copy a fixed amount of information from a disk or tape to RAM. This fixed amount might correspond to a single track and sector, for example. The unit used for transferring information is called a *physical block*. Any given file might be shorter or longer than one physical block. A shorter file will be given a single physical block, even though some of that block will remain empty. A longer file will need multiple blocks of storage, which may or may not be contiguous. When data is transferred into main memory, it may need to be reassembled from several different physical blocks on the disk. This reassembly occurs in a section of main memory set aside for this purpose, called a *buffer*.

When a file is stored on a disk (either magnetic or optical) its name, starting location (track, sector, and side), and length are written in a *directory*. Within the directory is also a list of available space on the disk. When a file is deleted, the file itself is not physically erased; its reference in the directory is simply deleted and the space it occupies is added to the list of available space. The user should be aware that information one thinks is deleted may remain on a disk for quite some time, until its space is written over by another file.

Files are usually organized in a hierarchical manner on a disk. Specialized files called *folders* or *directories* are created. These files contain no individual pieces of data but are used to organize other files. The main file, that contains all the others is called the *root*.

The whole structure can be pictorially represented by a diagram which looks like an upside-down tree so it is commonly called a *tree-structured file system*. See simple example at right.

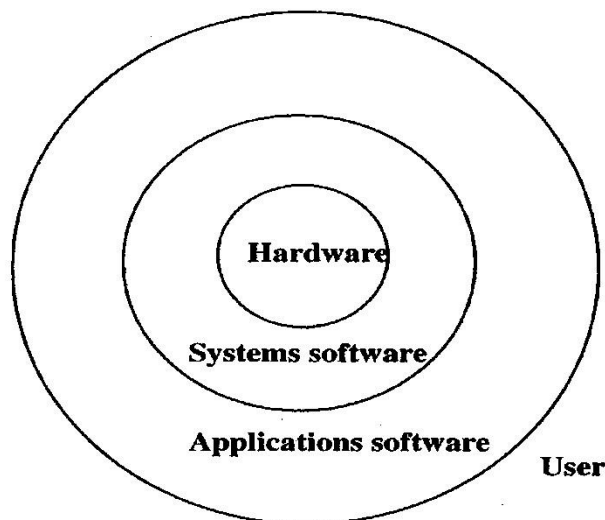


In this tree the root directory has three files and two directories in it. Each of the directories contains both files and directories and so on. Systems containing files organized this way normally contain methods of moving from one place in the tree to another. In graphical systems with folder icons, it is common to use a mouse to click on a folder. For example, if you clicked on the root folder, you would get a window showing file1, file2, file3, dir1, and dir2. If you then clicked on dir2 you would see a window with file5 and dir5 in it. If you clicked on folder dir5 you would see file9 and dir6 and, finally, if you clicked on dir6 you would see file10 and file11.

## 8.4 System Software

Software gives the computer the ability to go beyond the tasks built into the hardware and do a variety of tasks. Software is a general term for any program that runs on a computer. Hardware refers to the physical parts that make up a computer such as memory chips, disk drives or processors. Hardware is “hard” because it has a physical presence that one can touch; software has no physical presence apart from the hardware it runs on. There are two basic categories of software, systems software and applications software. Systems software consists of programs that control the running of the computer while applications software are programs that accomplish some goal for the user.

There are many complex details regarding the design of the hardware components of a computer that are important in the processing of data, such as memory size, operations available in this particular computer’s processor(s), location of a program in memory, etc. Yet the average computer user is often unaware of these details. The primary function of systems software is to act as an interface between the user and the hardware, setting up a “virtual computer” for the user. This virtual environment hides most of the technical details from the user, making the computer much more user friendly. The typical user does not need to know details such as memory size of the computer or the memory location of a given program. The systems software takes care of all this automatically. The systems software can be thought of as forming a series of insulating rings between the user and the hardware. Every piece of data or instruction given by the user works its way through these layers of programming until it is ready for execution by the hardware. When you type a command to your computer or click a mouse button, you are using the systems software on your machine.



There are several types of systems software, including translating programs, file management programs, and communications programs. We mentioned the translating programs when we first talked about high level languages in Chapter 3 and will discuss this further when we study low

level languages in Chapter 8. The remainder of this chapter will introduce the collection of programs called the operating system and some of the basics of network communications.

## 8.5 *Operating Systems*

The *operating system* of a computer is a collection of programs that define how the computer will interact with the outside world and that prepare a user's programs and data to interact with the hardware. The operating system sets up the virtual environment the user experiences, the “look and feel” of the computer, thus it is a very important program found in every computer system.

When a computer system is turned on or *booted up*, the operating system is automatically loaded into main memory and initialized. The part of the operating system that the user interacts with is called the shell. Most operating systems today, such as Windows, use a *graphical user interface*, or *GUI*, in which the tasks a user might wish to do are represented by pictures or pull down menus. The programs that actually execute the tasks are hidden from the user.

Some of the tasks the operating system does are:

- security management
- maintains accounting records
- manages external devices such as I/O devices and auxiliary storage
- allocates the CPU to users and runs programs
- manages programs and data in main memory

We will take a brief look at each of these.

### **Security**

Security is an issue, both on multi-user machines and on machines connected to others through a network. In either case it is vital that a user be able to control access to certain programs or data within the computer.

Most of today's operating systems maintain security by controlling access to the computer through a system of usernames and passwords. Only those users who know a valid username and password combination are given access to the system. Since computers can be programmed to go through a dictionary trying every word as a possible password, it is important for users to use some care in choosing a password that would not be easy for an outsider to guess or to crack.

Access to particular files is also controlled through the use of permissions. A user can set the permission status on a file so that others can read and execute a file (if it is a program), read only, or have no access. The operating system keeps lists of users and their status and checks these lists when a user asks for access to a given file.

## Accounting

Every computer has an internal clock. The operating system uses this clock to keep track of a variety of statistics, including each user's log on and log off time and the CPU run time of each program. These times can be useful for comparing the efficiency of different algorithms or for billing a user for their computer usage.

## I/O and Auxiliary Storage

The operating system oversees the passing of information between main memory and peripheral devices such as printers, keyboards, mice, disks, and flash media. The main task of the operating system when transferring information to or from input and output devices is to translate that information between its binary representation and the representation used on the I/O device.

The operating system also supervises and organizes the storage of information on disks and tapes. While sending information to a disk does not involve the translation of the information in the file, the operating system must locate available space on the disk, transfer the information, and record the location in the index for that disk. The operating system should make optimal use of the space available on the disk and transfer the information in as short a time as possible. Most operating systems maintain a list of available space and use various algorithms to find a good fit for a particular file. One method might be to simply store the file in the first available space that is big enough. Another might look for the best fit, the space most closely approximating the file's size. Some operating systems can split a file among multiple locations. When a file is deleted from a disk the space occupied by that file is added to the list of available space. There is no real point in actually deleting the contents; they will remain until that space is reused by another program. It is important for the user to realize that deleted files are not physically erased from most disks until their space is allocated to a new file. Thus files deleted by mistake can often be retrieved. And files containing sensitive data must be erased by special command. The easiest way to erase the contents of a disk is to *format* it. Formatting not only erases a disk, it also checks for defects and sets up a new index system. However, formatting erases the entire disk, so hard disks are rarely formatted once they contain information. If many short files have been added and then deleted from a disk the disk might become fragmented. A *fragmented disk* has available space but it is broken into small sections scattered all over, leaving few or no large segments of available space. There are utility programs called defragmenters that move programs and consolidate the available space.

On a multi-user system, a particular difficulty for the operating system is the allocation of peripheral devices among multiple users. Users can share main memory by having their programs in separate locations. They can also take turns using the CPU to run their programs. But devices such as printers or disk drives cannot be so easily shared. Once a user has begun a print job, for example, all other users must wait until that job is finished, for no one would want their information printed in the middle of someone else's. Many operating systems use a system of priorities to determine which programs are allocated which devices.

## CPU Allocation

Early computers had no difficulty determining when a program gets to use the CPU because they ran only one program at a time. While this is a simple way to run programs, it has obvious drawbacks when today's PC user expects to run several programs concurrently, (ie. Email, web browser, & a music player.) On a more capable computer that supports a multiple-user environment, if programs were processed on a first come, first served basis some users may have to wait quite a long time to get their program run. Most computer users today often wish to have multiple programs running at the same time. A second problem stems from the fact that the CPU is very efficient. The ordinary microcomputer can process at least a billion instructions per second, (1gigaFLOP is one billion floating point operations per second) while supercomputers can process over one petaFLOP, (peta=> $10^{15}$  operations per second!) On the other hand, devices such as printers may print only 100 characters per second and accessing something on a disk may take several milliseconds just to look up the address in the index, with several more milliseconds needed to spin the disk and move the read/write head to the proper location. In general, any operation that takes physical motion is much slower than the transfer of electronic bits within the machine. Thus machines that process one job at a time are said to be *I/O bound*, meaning that the CPU is idle much of the time while the processor is waiting for a peripheral device to complete its part of the job.

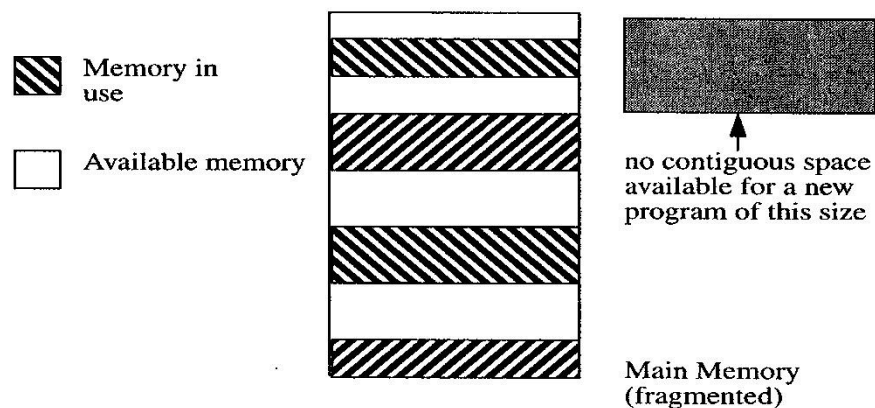
One solution to these problems is for users to share the CPU, allowing one job to process while the other is using the printer or disk drive. This is called *overlapped processing* and does solve the problem of slow peripheral equipment. However, it does not solve all the waiting problems. For example, suppose user A submits a program to be run that is very lengthy and will take a lot of CPU time. A second or two later user B submits a program. User B will still have a long wait while user A's program is processed.

A better solution is for the CPU to take turns working on each of the two programs, a system known as *multitasking* (or multiprogramming or timesharing). With multitasking, both programs are in main memory at the same time. When a program is submitted for processing it is placed on a list called a ready queue. When the processor is available it takes the program at the front of the *ready queue*, transfers its necessary information to the CU and starts processing. It continues processing for a given time interval, say one millisecond, called a *timeslice*. Then it takes a "snapshot" of the state of the process by recording the contents of all the registers in the CU. This allows it to return to the same point in the process later. After this it returns the process to the back of the ready queue and takes the next process from the front. This process is allocated its timeslice and then sent back to the ready queue. This way each program that is ready to run is given a turn. The user watching this process would not be able to tell that several programs are running at the same time because the timeslices are so short that before one can notice that the computer is not running one's own process the computer is back (most humans cannot perceive time spans of less than 40-50 milliseconds). Thus, to give the appearance of continuous processing it is important for the timeslice to be relatively short. Notice, however, that the time spent copying the contents of the CU is essentially wasted time. If the timeslice is too short, more time will be spent taking snapshots of the CU than is spent in actual processing. In most machines this switching time is less, however, than the time that the CPU is idle while processes wait for the use of printers or disks.

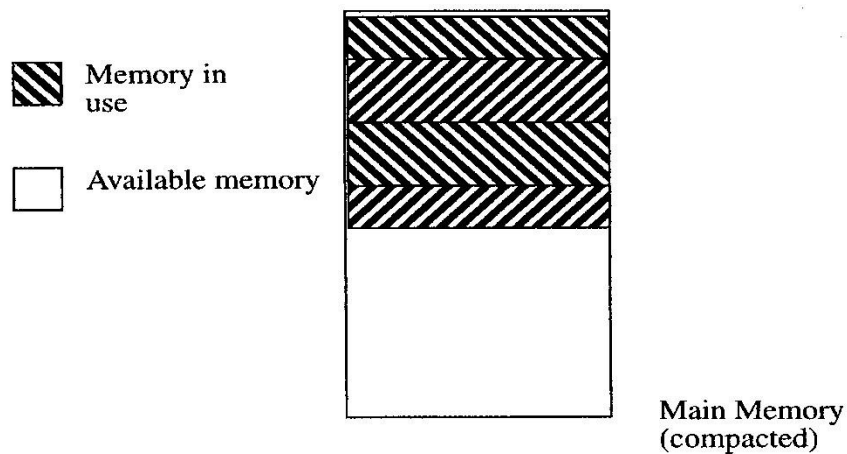
We have been considering operating systems for computers with a single processor or CPU. Many computers today have several processors attached to a single main memory. Jobs can be divided among these processors or a single job might be split, with several processors each working on one piece. This is known as *parallel processing* or *multiprocessing*. How an operating system might divide a single program among several processors is extremely complex and beyond the scope of this course.

## Memory Management

One problem an operating system might have when running several programs in a multitasking mode is that all the programs may not fit into main memory. As with storage on a disk, the operating system keeps a list of spaces available in main memory and attempts to fit new programs into space that is available. When programs complete their task or users log off, new spaces become available. This can lead to fragmentation of main memory just as it led to fragmentation on a disk. There may be plenty of space available, but it may be broken into small, noncontiguous pieces.

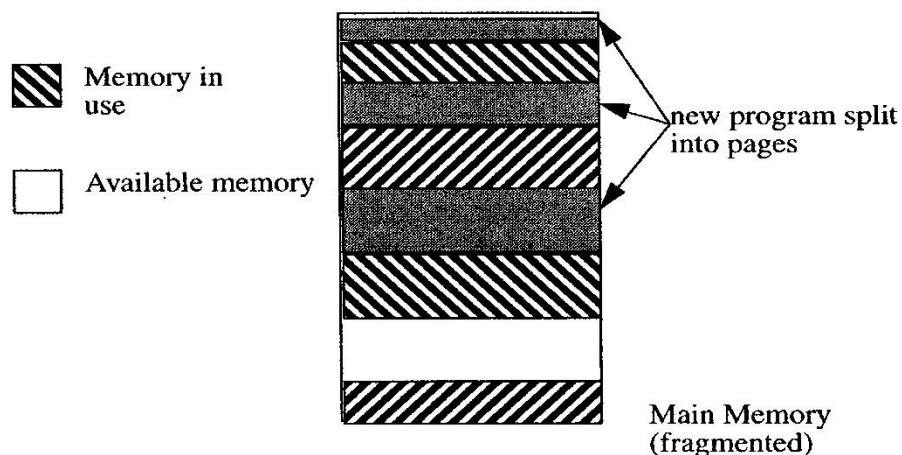


There are three solutions to this problem. The first is memory *compaction*. This involves running a compaction program that moves all the programs in main memory into one contiguous block, leaving the space available also as a contiguous block.



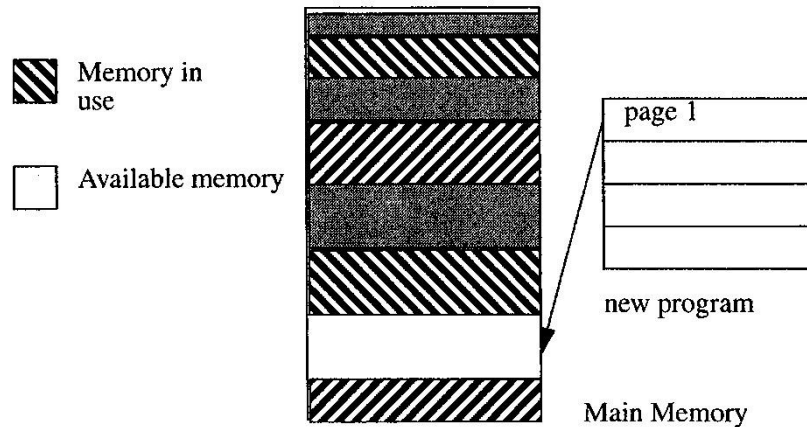
There are two drawbacks to this solution. First, running a compaction program takes time. The computer must change the addresses in all the jumps for each program that is being moved. While the compaction is taking place, the CPU is not getting anything else done. Second, once some of these jobs are completed the memory will once again be fragmented.

A second solution is to leave all the programs where they are and to break up new programs into smaller segments, called pages, that fit into the space available. This is known as “paging”. The operating system adds jumps between the different pages, so that the program counter knows where to find the next instruction when it reaches the end of a page.





It might also happen that there simply isn't enough memory available for a new program, even with paging. In this case it is possible for the computer to get the program started anyway, simply by bringing in only as many pages as it has room for, replacing these pages with new ones as needed. This is known as *swapping*.



Swapping slows down execution of a program since the computer will periodically have to retrieve new pages from much slower disk drives. However, it turns out that most programs do not jump from page to page very often. Most instructions follow from the previous one and when data is used from large sets, the next piece used is usually near the previous one. This principle of locality means that most programs can run for a long time using only a small portion of main memory. It is up to the operating system to determine how many pages need to be loaded at a given time, based on the nature of the program and the amount of space available. If a program jumps frequently between two pages, for example, it would be optimal for the operating system to note this and make sure both pages are loaded somewhere in memory.

Paging and swapping set up a memory system that is known as *virtual memory*. Instead of having the entirety of all programs that are running in contiguous blocks of main memory at once, they allow programs to be broken up and to have only a page or two in memory at once. This allows for many more processes to share the memory. Since paging and swapping are hidden from the user, the most the user will notice is a slight slowdown in processing speed.

### **Utilities Programs**

Utilities programs are programs that either enhance the functioning of the operating system or do various housekeeping tasks that are not included in a given operating system. Some tasks accomplished by utility programs include reorganizing storage on disks through defragmentation or compression, copying disk contents for back-up purposes, checking for viruses, or attempting to recover lost or damaged data on disks. While none of these are necessary to the operation of a computer they are all helpful for optimizing storage or for preserving data in the unlucky event of a viral infection or a system or hard disk crash (failure).

## Some Common Operating Systems

There are many operating systems in use throughout the world. Currently there are three companies that dominate the market in this area, Apple, Google, and Microsoft. All of these companies have produced operating systems for personal computers and for mobile devices.

A common operating system for personal computers is Windows, developed by Microsoft. Since the current versions of Windows also supports multitasking and paging, it can be used on workstations or in multi-user environments. Windows is GUI based, which means that most tasks can be accomplished by using a mouse to click on an icon or on a command in a menu. This makes Windows easy for the non-professional to use. While Windows is easy to use, it does take up a fair amount of memory space. Another disadvantage is that there is no convenient way to modify Windows or to fix bugs in the system since the source code is proprietary and unavailable to the public. And there are bugs. However, for the user who just wants to get a computer up and running with a minimum of fuss and to use standard applications such as word processing and spreadsheets, Windows is simple to install and use.

Unix is an older operating system initially developed in the 1970's that is commonly found on larger machines such as industrial workstations or multi-user mainframes, largely because of its reliability, flexibility and the quantity of software that it can run. In the past Unix was usually implemented as a command based system in which one enters short commands into a window on the screen, such as `cd` (change directory) to move into a subdirectory or `run` to execute a program, but now Unix offers both a graphical user and the command line interface. Unix has the strengths of openness and versatility and can run a wide variety of software including many open source packages. It can be implemented on most hardware platforms and is a good choice for networks that involve a variety of different processors.

Linux was introduced in 1991 as a Unix-like system for personal computers. It is available on the Internet as free software or from vendors at a nominal price. For computer professionals who are comfortable with Unix' commands and flexibility, Linux is a very popular PC operating system. One advantage of Linux, (and Unix) is that it is an open-source program. This means that one can read the code of the programs that make up Linux and modify this code to tailor the operating system to one's own needs. If a flaw, such as a security loophole, is located in the system you can look at the code and possibly find a solution yourself, rather than waiting for a company such as Microsoft to come out with a patch or a new version. Enhancements of the system are frequently posted on the Internet so other users can take advantage of them. There are also many companies that offer versions of Linux for sale and then also provide technical support to the users of their version of Linux. The Android operating system marketed by Google is based on Linux and specifically designed for mobile devices and has become a very popular OS on mobile phones and other mobile devices throughout the world.

Another very common operating system for personal computers is Mac OS X, developed by Apple. It offers all of the features of a typical operating system and is known for its innovative and easy to use GUI. Mac OS X is based on the Unix operating system and this has attracted some former Unix users to make the switch to OS X. Apple also created another operating system called iOS designed specifically for mobile devices such as their popular cell phone, the

iPhone, their tablet computer, the iPad, and their personal music player, the iPod. The easy to use GUI with its many features has made these devices and the iOS operating system more popular in the last few years.

One of the more recently developed operating system is Google's Chrome OS which is based on Linux and designed specifically to work well with web applications and applications installed on your local machine. It is also designed to run applications for the growing number of mobile devices that use the Android OS.

## **8.6     *Networks***

Computers can be used as stand-alone devices if they have the appropriate software to do the desired task, (ie. If your computer has a word processor, you could create a document.) More often though, a computer is connected to other computers and this collection of connected computers is called a "network". The network allows these computers to share each other's resources such as programs, data, and processing power.

Networks fall into two main categories, local area networks (LANS) and wide area networks (WANS). Local area networks are networks of computers limited to a single building or campus. Wide area networks are those that go beyond a local area, connecting computers from plant to plant, city to city, and across the world. LANS and WANS use different types of technology to connect the computers and generally share resources at different levels.

### **Types of Connections**

In a local area network, computers are often physically connected to each other using either twisted pair wiring, coaxial cables, or fiber optics. Twisted pair connections consist of two wires that transmit electrical signals. Much of the phone system in the US consists of twisted pair wires. Many wide area networks use the phone system to connect computers. Coaxial cable consists of an insulated copper wire. It is used for cable TV transmission. Coaxial cable is more expensive than twisted pair wiring but it is also faster and less prone to interference. Fiber optic cables are the fastest of all. Instead of transferring data via an electric current, fiber optic cables carry pulses of light generated by a laser. While fiber optics are much faster and less prone to interference than any of their wire counterparts, they are expensive to install.

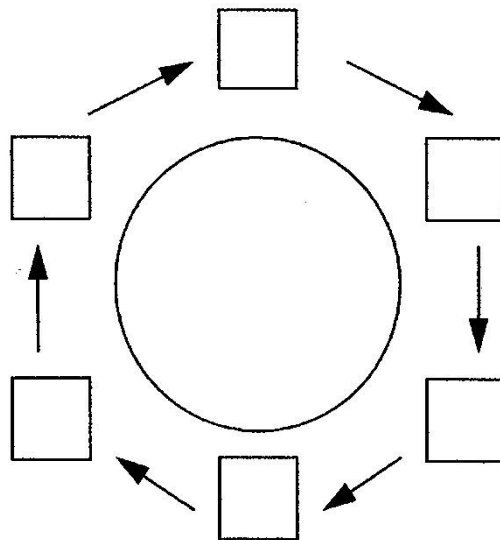
Computers can also be connected to each other without physical wires. Data can be transmitted via radio or microwaves. Radio transmission is often used in local area networks where physically wiring computers together would be inconvenient. A lab might be set up with one computer transmitting to all the others. Radio waves only work well in a local setting, as they are susceptible to interference from other radio transmitters or motors. Microwaves are the transmission method of choice for wide area networks. Microwaves use high frequency radio signals which travel easily through the earth's atmosphere. Microwaves only travel in straight lines, so relay towers must be set up at intervals, or satellites must be used to transmit over long distances.

Data that is communicated via the phone system is transmitted not as a series of electrical pulses, but as a continual electrical signal that forms a wave, much like a sound wave, varying in frequency and amplitude. This form of signal is known as an *analog* signal. Before transmission across the phone lines, data must be converted from its digital form to a corresponding analog form. The device that does this conversion, as well as converting incoming data from analog to digital, is called a *modem*. Each modem is capable of transmitting a certain number of bits per second (bps) and this number is its transmission speed. A typical modem should transmit about 56,000 bps, (56Kbps), while speeds for wireless transmission are much higher in the 600 to 1300 Mbps range, but require a nearby wireless access point with an equally fast or faster connection to the internet to take advantage of the speed. The speed of communication also depends on the capacity of the link between computers to carry bits. *Bandwidth* is used to designate the number of bits per second that can be sent over a given link. While twisted pair has a bandwidth up to 1 Gbps, (Gigabit), fiber-optic cables have been developed that can deliver hundreds of Gigabits per second and into the Terabit range although more typical is in the one to ten Gbps range.

Most personal computers come today with one or more ways to connect to other computers, (wireless network capability, a network card for wired connectivity, or an internal modem for connection via a phone line.) The network card is a standard feature and the other two are optional, depending on the user's needs and the type of computer.

## Network Topologies

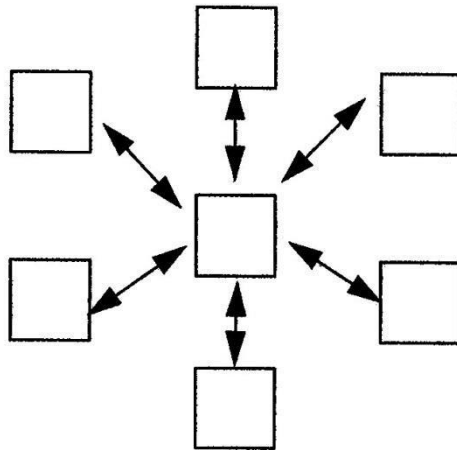
In a local area network there are three common ways the computers might be connected. The simplest is to connect each computer to two others, forming a ring of interconnected machines.



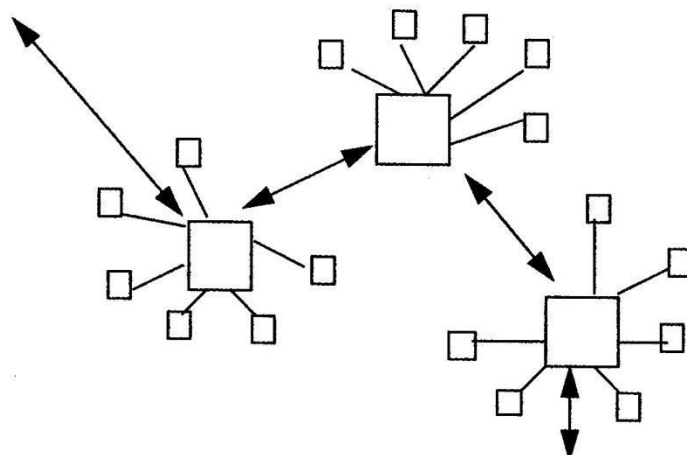
Messages are generally passed from one machine to the next in a single direction until the destination machine is reached. A similar topology, called a bus network works much the same

way, except the machines are all in a row and transmit messages in one direction at a time. Both these networks are easy to set up. Each machine is equal to the other. But they are slow in that only one message can be transmitted at a time and if one machine fails the entire network usually comes to a halt.

Another way to connect a network is to connect all machines to a central one in a star pattern.

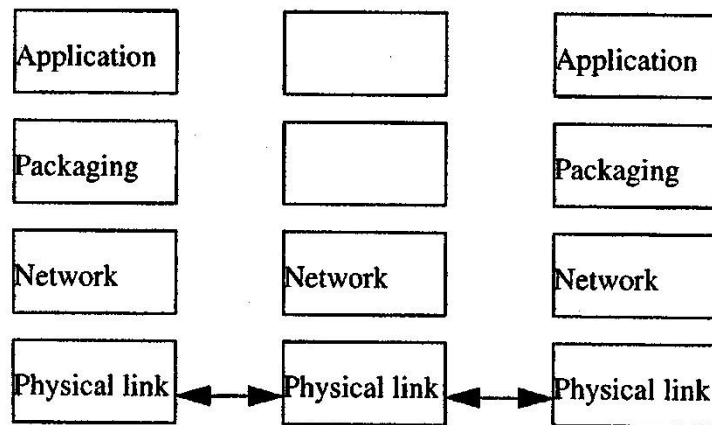


An advantage is that messages never have to pass through more than one intermediary machine and if a machine, other than the server, fails it does not affect the other machines. Star topologies are often used in a *client-server* network where the central computer acts as a host to the other computers, coordinating the transmission of data between them. Client-server networks are useful when several machines wish to access the same programs or data. Instead of each machine having its own copy, a single copy can be stored on the server and shared with the other computers as needed. The Internet is generally set up as a series of interconnected servers or hosts, each with their own set of clients.



## Protocols

A *protocol* is the set of rules governing transmission of data within a network. The protocol defines how the message must be addressed, which computers have the right to receive or transmit the message, how the message will be unpacked at its destination, and how errors in transmission are dealt with. The sender and the recipient, as well as any intermediary machines, must agree on a protocol before they can transmit messages to each other. A typical protocol works through a series of programs that prepare a message for transmission or reception.



A message is generated in one computer by some application, say a word processor. The message is then sent to the next level where it is packaged for transmission. The network layer is responsible for routing the data to its recipient. The link layer finally sends the message on to the next computer. Intermediary computers are consulted only down to the network layer. At this layer they check the address and send the message on. At the destination the message is received by the network layer, the packaging layer checks for errors in transmission and unpacks the message, and an application finally presents the message to the user.

The four-layer model described above is similar to FTP or file transfer protocol, commonly used on the Internet. Another, seven-layer collection of protocols used on the Internet is TCP/IP.

## 8.7 Conclusion

Although a computer only needs a CPU and main memory to function, most computers have input, output and auxiliary storage devices. Input devices are used to enter data and programs into the computer and include touch screens, keyboards, mice, scanners, digital cameras, microphones, and various other sensors. Output devices return data to the user and include monitors, printers, and speakers. Auxiliary storage provides long term, nonvolatile storage of programs and data and includes tapes, magnetic disks, and CDs.

The operating system provides the user with a collection of programs that overlay the hardware, making it possible to do many different kinds of tasks from word processing to video chats. We now know that computer hardware represents everything with patterns of 0s and 1s; that the CPU may be shared among many users or processes; that a user's program might be anywhere in memory, might be in several locations or might have only a small part in memory at a time; and that many operations we take for granted such as taking a square root or dividing one number by another require the use of programs built into the system while other operations such as addition have circuitry built in. The operating system hides all of this from us, giving us the illusion of being the only user, of having our entire program loaded into memory on command, of having a wide variety of operations available. It allows us to click on an icon or choose a simple command in order to have programs loaded, executed, printed or saved to a disk. This layer of programming between the user and the machine allows the user to think in much more abstract terms and ideas, such as decimal rather than binary numbers or words rather than ASCII codes, making the computer much more user friendly. The operating system sets up the environment in which the user works, making it the most important collection of programs within the computer.

Today, most computer users also go beyond the capacities of their personal computer, obtaining data, programs and sometimes even processing power from other computers to which they are connected through a network. Networks may be local or over a wide area. Local networks share the most resources among the connected computers. Wide area networks, such as the internet, generally share data only. To transfer data from one machine to another the two must agree on a protocol, or way of packaging the data.





# Chapter 9

## Data Manipulation

### 9.1 Data Manipulation: Boolean operations and Circuit Components

We have now seen how to encode and store data, both in the main memory of the computer and in auxiliary storage. This accounts for the computer's ability to add data or to move it from one location to another by copying bit patterns into different registers or from one track and sector on a disk to another. What we have yet to see is how that data can be changed by the computer. Recall that the computer need not have very many processing procedures built in. The ability to add and to compare two bit patterns is sufficient for a simple operational machine. In the next sections we will show how to design circuits for determining if two bit patterns are equivalent and for adding the contents of two registers.

The basis of the circuitry for all electronic computers is Boolean algebra. Boolean algebra has three logical operations, **NOT**, **AND**, and **OR**, that can be used in combination to build complex expressions that will evaluate to either true or false. This system fits well with the idea of using zeros and ones to represent and manipulate information in the computer, (zero for false and one for true.) Each of these three basic Boolean operations can be implemented as a single small circuit component called a *gate* which can be used to manipulate bits. Each of these gates has a place to connect one or more input bits to it and each has one output bit. Depending on the type of gate, (AND, OR, NOT), and the input(s) to it, the output of the gate will be either a zero or one. The circuitry of a computer is built up using many copies of these three building blocks to construct specific circuits to manipulate the data, (zeroes and ones) as desired by the computer builder.

**NOT:** a NOT gate has a single input and output. NOT simply reverses its input from 0 to 1 or 1 to 0. In other words, in terms of electric signals, if power is coming into a NOT gate the NOT acts like a switch and turns the power off. If no power is coming in, the NOT turns the power on and has power coming out. We can represent this in a table, called a *truth table*, as follows:

A	NOT A
1	0
0	1

Notice that NOT corresponds to the intuitive idea of what NOT means in a logical sentence. If A were to represent the statement "It is raining." then NOT A would represent "It is not raining." If

a statement is false, then NOT that statement is true and if a statement is true, then NOT that statement is false.

**AND:** AND receives two inputs. If both are 1 then the output is 1. Otherwise the output is 0.

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

In terms of electric switches, an AND gate requires power coming in on two lines in order to have power going out. In logical terms, AND corresponds to the formation of a compound statement with the word ‘and’. If we say “It is cold and it is raining.” this compound statement is only true when both of its parts are true. If either A, it is cold, or B, it is raining is false the entire statement is false.

**OR:** OR also receives two inputs. If either of the inputs is a 1 then the output is a 1. If both are 0 the output is a 0

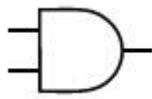
A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

An OR gate requires power from one of two, or both input wires for power to go out. OR corresponds to what we would expect in using or in a logical statement. “It is cold or it is raining.” is true either if it is cold, if it is raining, or if it is both cold and raining. (If we wish to limit the statement to one of A or B but not both we would use a different operation called exclusive or, or XOR. XOR is true when either A or B is true but not both.)

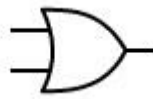
## 9.2 Boolean Expressions, Circuit Diagrams, and Truth Tables

We can combine several gates to get more complicated circuits. We have three ways of representing these combinations. The first is as a Boolean expression, such as A AND NOT B. We will simplify our writing of expressions by using AB to designate A AND B,  $A + B$  to designate A OR B, and  $A'$  for NOT A. Thus the expression (A AND NOT B) OR B would be written as  $AB' + B$ . When creating or evaluating combinations of the logical operators, NOT, AND and OR, it is important to realize that these logical operators have an order of precedence. Similar to the rules of high school algebra, what is in parentheses is evaluated first. Here is the order of operator precedence for Boolean Algebra. 1. ( ) 2. NOT 3. AND 4. OR

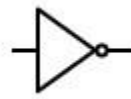
The second way to represent a combination of gates is to use a circuit diagram. On a circuit diagram we represent the logical operations of AND, OR, and NOT using the following symbols.



AND gate

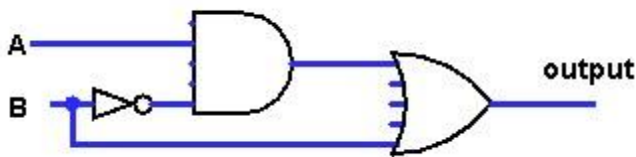


OR gate



NOT gate

The expression  $AB' + B$  would be diagramed as:



A third way of describing this combination of gates, (circuit), is to use a truth table:

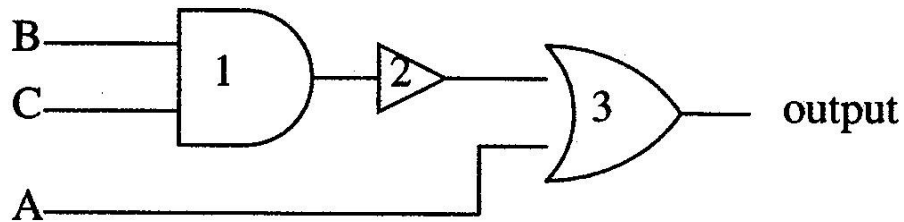
A	B	B'	AB'	AB'+B
0	0	1	0	0
0	1	0	0	1
1	0	1	1	1
1	1	0	0	1

We can derive each of these three ways of representing a combination of gates, expression, truth table, or circuit diagram from any of the others. To find the corresponding truth table for an expression we make a column for each letter, with enough rows to allow for all possible combination of inputs. If we have one letter in the expression we need only  $2^1$  or 2 rows since our only inputs are 1 and 0. If we have two letters in the expression we need  $2^2$  or 4 rows since we can have inputs of 0 and 0, 0 and 1, 1 and 0, or 1 and 1.

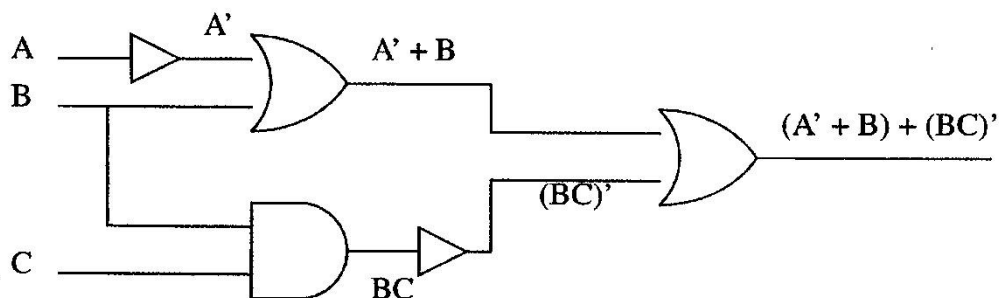
With three letters in the expression we need  $2^3$  or 8 rows, since there are 8 different combinations. The next step is to begin with the simplest (or innermost if there are parentheses) sub-expressions, making a column for each sub-expression until the entire expression is reached. Suppose we are given an expression:  $A + (BC)'$ . We need one column for each letter in the expression and one column for each subexpression. Starting with the parentheses in this case we have the subexpression, BC. The next would be  $(BC)'$ , and the third expression would be the entire expression. Our truth table will need 8 rows and 6 columns (for A, B, C, BC,  $(BC)'$ , and  $A + (BC)'$ ).

A	B	C	BC	(BC)'	A+(BC)'
0	0	0	0	1	1
0	0	1	0	1	1
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	0	1	1
1	0	1	0	1	1
1	1	0	0	1	1
1	1	1	1	0	1

To draw a circuit diagram from an expression, we follow a similar procedure. Begin drawing with the innermost part of the expression, then add other gates as needed. Thus we begin with an AND gate for BC, add a NOT gate to the end, then add an OR gate.



Given a circuit diagram, we can find the corresponding expression, and from there the corresponding truth table, simply by labeling each wire on the diagram with the expression to that point, starting on the left. To make the truth table use the expression for the output of each gate to label one of the columns in the truth table, along with a column for each letter.



## Sum of Products

What remains, is to see how to find the expression if we are given the truth table only. There is a simple method to find this called the sum-of-products method. The algorithm for this method is as follows:

1. For each row with a 1 in the final column, AND together the letters with a 1 in their column and the negation of the letters with a 0 in their column
2. Connect these AND groups with ORs (+)

Thus for the truth table:

A	B	?
0	0	1
0	1	1
1	0	0
1	1	1

we first produce AND clusters (products) for rows 1, 2, and 4:  $A'B'$ ,  $A'B$ , and  $AB$ . We ignore row 3 since it does not have a 1 in the final column. We now connect these with ORs (sums) getting the expression  $A'B' + A'B + AB$ .

## 9.3 Simplifying Boolean Expressions

Unfortunately, we often get rather unwieldy expressions using this method. Thus it is important to have some ways to simplify an expression. We use the following laws of Boolean algebra for simplifying expressions:

1. Commutative law

$$X + Y = Y + X$$

$$XY = YX$$

3. Idempotency law

$$X + X = X$$

$$XX = X$$

5. DeMorgan's law

$$X'Y' = (X + Y)'$$

$$X' + Y' = (XY)'$$

2. Distributive law

$$XY + XZ = X(Y + Z)$$

$$X + YZ = (X + Y)(X + Z)$$

4. Double Negation

$$(X')' = X$$

6. Identities

$$X0 = 0, \quad X + 0 = X$$

$$X1 = X, \quad X + 1 = 1$$

$$XX' = 0, \quad X + X' = 1$$

\*\*\* Note that in the Boolean algebra laws stated above, X and Y can be compound terms such as  $AB'$ .

To simplify the expression we derived from the given truth table,  $A'B' + A'B + AB$ , we first look for two terms that have one or more letters in common and use the distributive law to extract this letter, getting:  $A'(B' + B) + AB$ . Now note that  $B' + B = 1$  and  $A'1 = A'$ , so we can reduce the expression to:  $A' + AB$ . We now distribute the  $A'$ , resulting in  $(A' + A)(A' + B)$  and again note that  $A' + A = 1$ , so our final simplified expression is  $A' + B$ . Below is the complete, step by step simplification, showing the Boolean algebra rule used to justify each step.

Expression	Rule Used in the simplification step	
$A'B' + A'B + AB$	original expression	
$A'(B' + B) + AB$	Distributive law	$XY + XZ = X(Y + Z)$
$A'1 + AB$	Identity	$X + X' = 1$
$A' + AB$	Identity	$X1 = X$
$(A' + A)(A' + B)$	Distributive law	$X + YZ = (X + Y)(X + Z)$
$1(A' + B)$	Identity	$X + X' = 1$
$(A' + B)1$	Commutative law	$XY = YX$
$A' + B$	Identity	$X1 = X$

We wish to simplify an expression before we draw a circuit diagram because the more complicated a circuit is the more space it takes up on the motherboard and the slower it runs. We want the simplest expression possible in order to have the most efficient circuit. Every gate in a circuit slows down the execution time, so the best circuit is the one with the fewest gates that does the task.

## 9.4 Designing Circuits

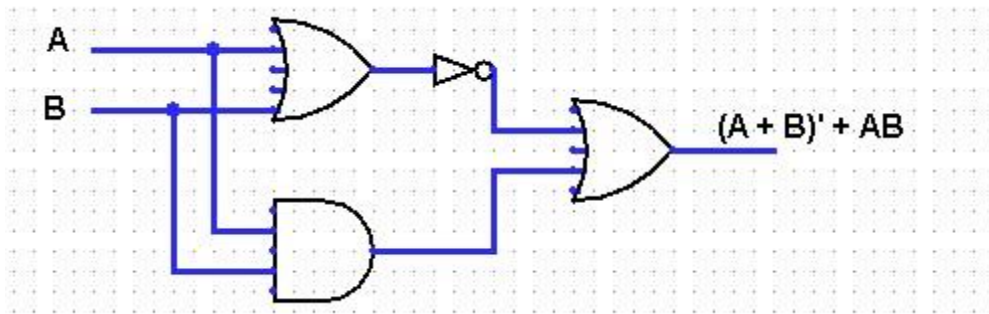
When designing a circuit, a truth table is often used as the starting point as a way to describe what task you want your circuit to perform. Once the truth table has been constructed, you can derive a Boolean expression from the truth table, (using the sum-of-products method), simplify the expression, and then build the circuit from the simplified expression.

### Circuits for Equivalence and Addition

To design a circuit for equivalence, we need to compare the contents of two registers. We do this bit by bit, starting on the right and moving to the left. Our circuit simply needs to take two bits as input and determine if they are both 0s or both 1s. The truth table for equivalence would be:

A	B	A=B
0	0	1
0	1	0
1	0	0
1	1	1

Using the sum-of-products method, we derive the expression  $A'B' + AB$  from this table. We can reduce this expression by one gate using DeMorgan's law that  $X'Y' = (X + Y)'$ . The corresponding circuit is:



Developing a circuit for addition is slightly more complicated. Recall that the rules for binary addition are:

$$\begin{array}{ll} 0 + 0 = 0 & 0 + 1 = 1 \\ 1 + 0 = 1 & 1 + 1 = 10 \end{array}$$

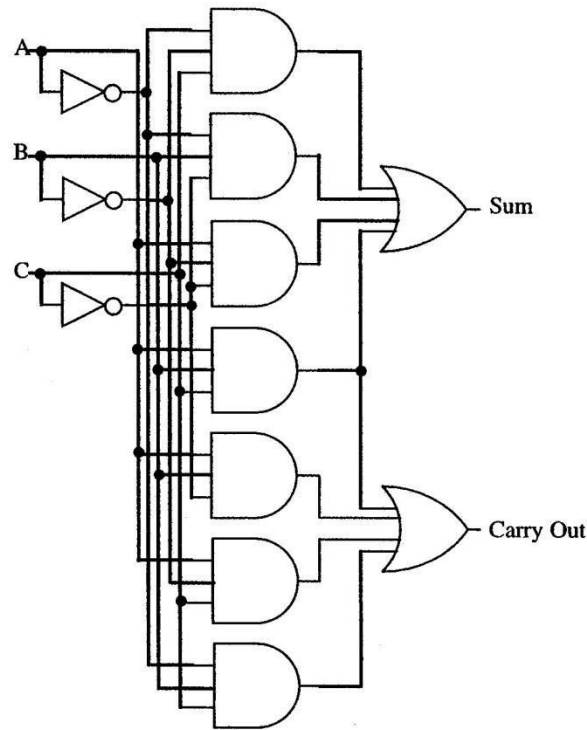
If we did not have the carried 1 in the last sum this would correspond to the following truth table:

A	B	A "PLUS" B
0	0	0
0	1	1
1	0	1
1	1	0

Notice that the final column of this table is the negation of the final column of our table for equivalence. The circuit for this would be the same as that for equivalence, with a final not gate on the end. This circuit is called a half-adder, since it carries out half of what we need for addition. To produce a full adder for two registers, we note that we really need to add two bits at a time, starting at the right, plus any carry that might come from the previous addition. Thus an adder needs to deal with three inputs, rather than two. It will also produce two outputs, one for the sum and another for the carry. Thus a truth table for full addition would be the following:

A	B	Carry In (C)	Sum	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

This circuit is significantly more complicated: The sum of products forms for Sum and Carry Out are  $A'B'C + A'BC' + AB'C' + ABC$  for Sum and  $A'BC + AB'C + ABC' + ABC$  for Carry Out. Note that both of these share the term  $ABC$ . We can combine these two into a circuit with seven AND gates and two outputs, as seen below. This represents the unsimplified version of this circuit. Using our Boolean algebra laws we could design a simpler circuit, but this one shows how we can derive the circuit directly from the truth table.



Notice that the output from Carry Out would be the input C for the next pair of bits. C would be 0 for the first, rightmost, pair.

These two circuits form the basis of the central processing unit, or CPU. As previously stated, we can build more complicated operations through the use of programs.

## 9.5 Conclusion

At the heart of computing is the storage and manipulation of data. This chapter focused on the manipulation of data. To actually change the data we need circuits that can add and compare. The capability of electronic computers is based on the three simple Boolean logic operations of AND, OR, and NOT. Using gates that can perform only these three operations, complex circuitry can be designed and created to manipulate binary data in many ways.

To manipulate or change data, circuits are used that will take one or two pieces of data, encoded as binary bit patterns as input and return a different binary pattern as output. Such circuits can be represented in three ways, as a truth table, as a Boolean expression, or as a circuit diagram.



# *Chapter 10*

## *Social and Legal Issues*

### *10.1 Privacy*

With the increasing use of computer networks and databases we now have access to far more information than ever before. We also have access to new avenues of communication and information dissemination. However, this does not come without a cost and one of the primary costs of increasing computerization of both the workplace and society in general has been an erosion of personal privacy.

#### **Surveillance**

Workers engaged in data entry or customer assistance are often monitored for the number of keystrokes or customer calls they produce per hour. Other professions such as nursing or package delivery require workers to log in each time they arrive at a destination or deal with a patient or customer. While such monitoring may increase productivity, it also can erode the relationship between workers and supervisors. Work at a pace set by a machine may cause added levels of employee stress and result in less accuracy, or diminished courtesy in serving customers.

Many now have access to e-mail, the Internet, and a variety of on-line databases at their workplace. Communication, both within and beyond the workplace, is often computer mediated. Workers cannot assume that any of this communication is private. Employers, advertisers, and government agencies all monitor electronic communication in ways that are often invisible to the user. In the United States, approximately three-fourths of the major companies monitor employee email and/or internet usage. Courts have generally sided with employers in disputes over such monitoring and employees have been fired for improper computer use at several companies, ranging from Xerox, Salomon Brothers Smith Barney, and Dow Chemical, to the New York Times. A recent survey by the American Management Association found that 45% of its member companies had disciplined workers for improper or personal use of company e-mail and 42% had disciplined workers for improper use of the internet.

Personal internet use may also be monitored by the web sites visited, advertisers, and extraneous third parties. Cookies, data files stored on a user's hard drive by a web site visited which identify internet users and collect personal information, are widely used on commercial web sites. Cookies are used to store such things as passwords, sites visited, and transactions at a given site. They allow businesses to personalize their web site to match a customer's preferences or interests, they facilitate on-line ordering, and they allow businesses to compile data on visitors to their web sites. Cookies might be downloaded by the owner of a site visited or by advertisement banners on the site and are generally created without the user's consent or knowledge. While

cookies cannot harm a user's computer, many users are uncomfortable at the thought of information about them being collected and stored without their knowledge. Many internet browsers now offer an option that allows users to disable cookies.

Other techniques called 'data warehousing' and 'data mining' make it possible to put together data from different sources and extract hidden information or patterns. Data that was gathered by separate sites can be combined creating a whole that is much more revealing than any of the individual parts. While any one piece of data may have little effect on a person's privacy, data collected from a variety of sites could be more revealing. Because of the power of data mining, it is important that separate data files on an individual be kept as difficult to link as possible. In the past, social security numbers were frequently used as key fields for personal records. The prevalent use of a common key simplifies the linking of data files, increasing the potential of unauthorized access to personal data. A social security number, perhaps on a student id in a stolen wallet, also facilitates identity theft, one of the fastest growing crimes in the US.

### **Database Security**

Increasingly large amounts of data are stored in both public and private databases. Data on an ordinary citizen might be held in databanks at that individual's workplace or school, banks, insurance companies, credit bureaus, medical care providers, credit card companies, schools, electric and telephone companies, stores frequently shopped at, and even restaurants patronized. Data on private citizens is also held by the IRS, Social Security administration, police, FBI, welfare agencies, Department of Motor Vehicles, and other government agencies. With the advent of the internet, much of this data can be accessed, both officially and unofficially, making it quite easy to put together a rather complete dossier on an individual's life.

In the US the laws protecting the collection and exchange of personal data are weak. The European Union agreed in 1995 to a Data Protection Act that provides the following safeguards:

1. Data can be obtained only by lawful means, with the subject's knowledge and consent.
2. Data collected must be relevant, not excessive, and for the purpose stated. Provisions must be made to keep it accurate and up to date.
3. The purpose for the data collection must be disclosed and the data cannot be used for any other purpose. Data should not be kept for longer than is necessary for the original purpose.
4. The data may not be disclosed or sold to others without the subject's consent.
5. Data collectors must take precautions against theft, loss, deliberate alteration, and unauthorized access.
6. Data collection policies must be openly stated. Subjects should have a readily available means of discovering the existence and nature of any personal data collected.
7. All data is open to inspection by the subject without excessive delay or expense. Subjects have the right to the rectification or erasure of incorrect data.

These safeguards provide reasonable protection for the individual and have been agreed upon in order to safeguard the flow of information between member states of the European Union.

United States privacy laws are not so simple. The Federal Privacy Act of 1974 gives each citizen the right to discover, correct and limit the dissemination of personal information held by the federal government. The act also prohibits the government from selling an individual's name and address. Many states have similar laws restricting the use of state held information.

There is no one guideline for privately held databases. In general, US law has favored business interests over private interests. Few US regulations limit the collection or dissemination of personal data. A variety of acts cover specific types of data. These include the Fair Credit Reporting Act of 1970, which regulates the disclosure of personal information by consumer credit reporting services. It requires such services to adopt 'reasonable procedures' to ensure the accuracy of their credit reports. However, no restrictions are placed on the type or amount of data that can be collected. The Family Education Rights and Privacy Act of 1974 protects the confidentiality of student records and allows students access to their records. The Video Privacy Protection Act of 1988 prohibits the disclosure of video customer rental records, but not customers' names and addresses.

The European Union signed an agreement in 2007 with the United States over privacy law. The EU's guidelines prohibit the transfer of personally identifiable data to third countries that do not provide an 'adequate' level of privacy protection. The signing of this agreement was significant in that it facilitates the transfer of customer data between the US and Europe.

## ***10.2 Security***

Databanks, computer programs, and computer systems themselves are all vulnerable to damage. The cause of damage may be accidental or malicious. In either event it is important that every computer user and owner take precautions to safeguard the integrity of both programs and data. In this section we will first examine the most common security problems and then consider what a user can do to prevent or alleviate damage.

### **Accidental Damage**

Accidental damage to computer systems or data has two general sources. Damage may be caused by natural or man-made disasters such as a lengthy power outage, fire, flood, tornado, hurricane, earthquake, war, or terrorist attack. Any business relying on its computer for crucial operations must be sure to have a disaster plan that takes into account those natural and man-made disasters possible in one's geographical location.

Other accidental damage may be caused through human error or carelessness. These include software bugs, or errors in programming that cause a program to function in ways other than were intended. Most large programs do contain bugs, though few of them cause major problems. A second common error source is simply human error. This includes the entry of incorrect data or the failure to follow procedures properly when using a program.

## **Malicious Damage**

Much damage is caused to both computer systems and the data and programs stored on them through intentional sabotage. Such sabotage includes theft or vandalism of equipment, unauthorized tampering with data or programs, and the introduction of harmful programs.

Computer viruses and worms have the capacity to wreak havoc on entire networks of computers. A virus is a program that includes instructions to destroy or corrupt data on the host system. A *worm* is a computer virus that can spread over a network. A *bacteria* is a program that causes no specific damage on its own, yet is programmed to make copies of itself, eventually using up all available disk and memory space and thus, overwhelming the system and causing it to crash. A *Trojan horse* is a program for some legitimate use, such as a game, that contains a virus or worm within it, but is also designed to steal private information. Once you run the host program, the virus or worm is copied onto your hard drive, where it may do immediate damage, or where it may reside until activated by some other program or on a specific date.

## **Security Measures**

There are several options for securing a computer system and its data. The first and most important is to make frequent back-up copies of data and programs as a precaution.

Unauthorized access to or tampering with data or programs can be discouraged in several ways. First, passwords should never be written down and they should be frequently changed. Words that are in the dictionary or common names should not be used as passwords as they are easily guessed or can be found by programs written to try dictionary words in sequence. Passwords with numbers and letters intermixed are the most secure.

According to a Michigan State survey, 75% to 80% of deliberate sabotage of data or programs, or use of computers in illegal activities such as fraud or embezzlement, is an inside job, perpetrated by disgruntled employees. Businesses need to control access to files and programs, limiting it to those who have a legitimate need in order to do their job. Businesses also should consider installing auditing programs that track programs and data files accessed and record changes and transactions. Finally, it is important that employees be well screened at hiring, well treated, and, if terminated, that they be escorted and denied access to computer systems from the moment of notification.

Hackers and crackers attempting to access files or programs from outside, whether out of curiosity or with malicious intent, can be discouraged through the use of a *firewall*. A firewall is a set of programs that add further access controls to those already in the operating system. They are designed to monitor and control access to a computer system from any networks to which the system is connected. Good password control and frequent auditing of system use also discourage unauthorized access from outsiders.

Anti-virus programs should be installed on all computer systems. Due to the continual production of viruses, the anti-virus software needs to be updated regularly, since they only work for viruses known at the time of installation. Besides using anti-virus software, one should also be wary of using disks from unknown sources or of downloading executable programs from

bulletin boards, social media sites, or other internet sources. Opening e-mail messages from persons unknown to you can be another source of viruses.

Finally, all computer systems should be covered with a disaster recovery plan. This plan should include provisions for backing up data and storing it both locally and off-site, for moving operations to a new location if needed, and for ways to alert or train necessary personnel.

### ***10.3 Ownership and Copyright Laws***

In July, 2000, Stephen King surprised the publishing world by publishing his latest novel himself over the internet. In the same month, a federal judge ruled against Napster, the internet site that allows users to exchange MP3 music files. Computer technology is changing the way information and ideas are distributed and raises questions regarding intellectual property rights. Intellectual property refers to any product of the mind, such as a book, music, an idea, or a procedure. Intellectual property has been protected in the past by patent law, for procedures or inventions, or copyright law, for books, music, or software. Current laws, designed primarily to protect the ownership of physical objects, are difficult to enforce in a networked age. Patents are granted for inventions and methods of doing something. There has been a recent controversy over the patenting of procedures on the internet, such as one click shopping, patented by Amazon.com. Patents have also been granted to software companies, giving them the sole rights to certain methods used in their programs. Patents work in favor of large corporations who have the legal staff to apply for patents on every idea they can. While the software industry argues that patents stimulate creativity, too many patents of rather common sense ways of doing things have the opposite effect, effectively curtailing the use of good ideas in new settings, thus slowing innovation in software development.

Copyrights prohibit the copying of intellectual property such as books, music, pictures, and software. While patents protect an idea or method itself, copyrights protect the expression of the idea in a particular form. Incorporating someone else's pictures, sound clips, or drawings on a web site is probably an infringement of their copyright. So is the downloading of books or music from the web.

An equally serious violation of copyright is the pirating, or copying of proprietary software, that is, software not in the public domain, or labeled as freeware or open source. When you purchase software, what you have actually purchased is a license that authorizes you to copy that software for limited use. Software licenses come in a variety of forms. The software you purchased may be licensed to you as a user, to be run on a particular machine or at a particular site, or to be run anywhere by anyone, but not copied. Multiple use licenses, purchased at a school or business, generally allow the licensee to run a specific number of copies of the software. Thus, it may be illegal for you to use a copy of software purchased for a business computer at home, or to give software you have purchased to a friend, even if you no longer intend to use it. One should also be wary of purchasing software in countries which do not enforce international copyright laws. It is estimated that up to 90% of the software sold in these countries has been illegally copied, or pirated, and is sold without a license.

## ***10.4 The Computerization of Society***

The increasing use of computers is affecting many parts of our society. Computers present both opportunities and dangers to both individuals and society as a whole. In this section we will simply point out several areas in which computers are rapidly changing our society, with mixed results.

Computers have changed the nature of many jobs. While they have taken over the routine tasks of many workers, they have also spawned a new set of jobs, such as that of data entry clerk, that are equally boring and routine. Computers have made possible a global market, for good and for ill. They have also changed the nature of many jobs, setting a faster pace, replacing face to face contact with e-contact, and raising the value of those things that can be quantified and digitized over less tangible concepts and concerns. Although computers were expected to shorten the work day or work week, they have often had the opposite result. Many workers complain that the time needed simply to answer e-mail adds an hour or two to their day. In an age of easy word processing or report generation, many workers have found that they are expected to generate more reports, or more drafts than might have previously been expected. And while the use of the proper software may significantly increase a worker's productivity, one must also factor in the time needed to learn that software and the problems of down time when the hardware or software malfunctions.

There is still a gap between those who have access to computers and those who do not. While many have prophesied that the computer will become a great leveler, giving all access to the same information and facilitating democratic processes, this vision has yet to be realized. The poorer the school district, the less likely the students are to have access to computers or the internet.

Computers are changing our use of leisure time as more and more time is devoted to surfing the web, or to chatting via e-mail rather than direct contact. While the computer gives us access to new forms of entertainment, what are we giving up in favor of screen time? Studies have shown that increased internet usage is correlated with increased depression and isolation.

Computers and the internet have placed much valuable information within easy access. The down side of this is that we may find the sheer volume of information overwhelming. It can be quite difficult to sort the accurate or meaningful information out of this information glut, a task that used to be done by editors and review committees.

Computers are here to stay. For most of you, they will play an increasingly important role in your lives. However, it is important that each of us use them as wisely as possible. We must continually ask ourselves what we are gaining and what we are losing when we choose to use a computer for a particular application or task. I hope that the knowledge you have gained in this course will help you make wise and informed choices as to when and how to use computer technology.

# *Appendix A*

## *Visual Basic Reference Guide*

### **Objects and Events**

An object is a form or anything on a form (buttons, textboxes, pictureboxes, labels, etc.). Every object has a name and associated properties. Some objects (buttons especially) have an event associated with them, that occurs when you click on the object. For each event you write a short program or procedure sometimes referred to as a subroutine.

Rename all objects so that the object name is meaningful. Keep a list of the object names associated with everything on your form. You will need the names of Textboxes and Picture boxes to do input and output in your program. You cannot use object names as variable names in your program. All names must be unique to one object or one variable and cannot be reserved words.

### **Variables**

*Names:* variables can be named with a letter or a string.

*Types:* every variable has an associated type, which tells the computer how to store the data in that variable. Common types are Integer, String, Single (single precision, or short floating point), Double (double precision, or long floating point), Long (long integer), and Boolean (True or False).

*Declarations:*

general form:       Dim variable As type = initialValue  
for example:       Dim price As Single = 0.0

More than one variable may be declared on a line but each must have its type specified:

Dim firstName As String = "", grade As Integer = 0

Arrays are declared as follows: Dim monthsOfTheYear(12) As String

Always declare all variables. Including the line Option Explicit at the beginning of a program forces you to declare all variables.

*Scope:* variables declared inside a subroutine (between Private Sub and End Sub) can only be accessed by the code for that particular button. Variables declared at the top of a form code page (not inside of a subroutine) are available to all subroutines or sections of code on that form. Variables global to all forms in a project are declared using Public rather than Dim and must be declared in a separate code Module. You can add a code module to a VB project from the menu, choosing Project, Add Module and then in the module type in the global variable declaration. Here is a sample declaration for a global variable:   Public Sum As Integer

A code module can also contain other code you may want that is not associated with a particular button.

*Assignment:* variables receive values in assignment statements of the form:

age = 19	'integer variable
firstName = "Joseph"	' String variable
n = txtNumberbox.Text	'from a text box
lastName = InputBox("Enter your last name")	'InputBox function
found = True	'Boolean variable
worker(N) = "Harlo Curran"	' use of String array
grandmaBirthday = #12/16/1941#	'assigning to a Date variable

*Order of Precedence:* ^, then \* or /, then + or -

When deciding on the precedence, as in algebra, parentheses may be used to assist in determining the order of operations. The logic within parentheses is evaluated first before combining with other terms.

## **Numeric and String Functions**

*Numeric Functions:* these can be used to directly assign values to a variable or as part of an expression. x is assumed to be a variable declared as Single in the following section.

Math.Sqrt(x) calculates the square root of x

Abs(x) calculates the absolute value of x

Int(x) truncates the value of x, yielding an integer

Here are some code samples that use numeric functions:

```
MyAnswer = Math.Sqrt(x) + 3*x - 5
```

```
outResults.AppendText(Math.Sqrt(x) + 7)
```

```
MyAnswer = Abs(x)
```

```
If (Int(x) = 13) then
```

```
    outResults.AppendText("This is your lucky day!")
```

```
End If
```



## String Functions

There are also several built in functions for manipulating strings. String functions can appear in output statements or their results can be assigned to a string variable.

Left, Right, and Substring return a substring from the left side, right side, or middle of a string.

sample use of the function: `outResults.Text = (Strings.Left("Saint John's", 7))`  
'note the use of the Strings prefix

results produced: Saint J

sample use of the function: `outResults.Text = (Strings.Right("Saint Ben's", 5))`  
'note the use of the Strings prefix

results produced: Ben's

sample use of the function: `outResults.Text = ("St. Joseph".Substring(4, 4))`  
'start at index 4 and take 4 characters  
'note that the index starts at 0

'note the absence of Strings when using Substring  
results produced: Jose

These functions can also be used with String variables, instead of the literal strings used above. Here is a segment of code that demonstrates using the Left, Right, and Substring functions with variables.

sample use of the functions:

```
Dim yourTown As String = "Smallville"  
outResults.AppendText(Strings.Left(yourTown, 5) & vbNewLine &  
    Strings.Right(yourTown, 8) & vbNewLine &  
    yourTown.Substring(1, 4))
```

results produced: Small  
allville  
mall

Some String functions return a number rather than a string. The IndexOf function searches a string for a character or substring and returns the index of the first position it is found. If it is not found, -1 is returned. This function takes two parameters, a string to search for and the index to start the search at.

sample use of the function:

```
Dim positionOfComma As Integer = 0, startPosition as Integer = 0
```

‘look for a comma and return the position in the string where it was found  
‘ \*\* remember that the index of the first character is 0

```
positionOfComma = "Avon, MN".IndexOf(",",startPosition)
```

```
outResults.Text = "A comma was found at index " & positionOfComma
```

results produced: A comma was found at index 4

Some String functions are used as if they were properties of a String object, much like the Text property is used for text boxes. You can access the value of these attributes by referring to the property using the name of the String variable and the property name. You can assign the property value to variables, use it in an expression, or display it an output box.

The Length function returns the length of a string, including spaces.

sample use of the function:

```
Dim yourTown as String = "Avon" outResults.AppendText("The length of the name of your  
town is: " & yourTown.Length)
```

results produced: The length of the name of your town is: 4

The Trim function returns a copy of the original string with all of the spaces removed from the beginning and end of the String. The Trim function does not change the original String.

sample use of the function:

```
Dim yourTown As String = " Avon Springs "
```

```
outResults.AppendText("Your town is:" & yourTown.Trim & ", MN!")
```

results produced: Your town is:Avon Springs, MN!

The ToUpper function returns a copy of the String with all of the characters in UPPER CASE. It does not change the original String. (There is a similar function, ToLower.)

sample use of the function:

```
Dim yourTown As String = "Avon Springs"  
outResults.AppendText("Your town is: " & yourTown.ToUpper)
```

results produced: Your town is: AVON SPRINGS

Some String functions return a Boolean value, either True or False. One that you may wish to use is the Contains function. This function takes a target string as a parameter and returns True if the original string contains the target string and False if it does not contain it.

sample use of the function:

```
Dim yourTown As String = "Avon Springs"
outResults.AppendText("It is " & yourTown.Contains("ring") &
    " that " & yourTown & " contains the word 'ring'.")
```

results produced: It is True that Avon Springs contains the word 'ring'.

## Control Structures: Conditions

*If Then:* Conditions can use =, <, >, <=, >=, or <> (not equal)

Simple conditions: The next two If statements are identical in meaning. The first one is a single line If statement. The second If statement has the same logic, but uses a block style that requires the use of an End If statement.

1. If age >= 16 then outResults.AppendText("You may apply for a driver's license.")
2. If age >= 16 then  
    outResults.AppendText("You may apply for a driver's license.")  
End If

Compound conditions can be created by joining conditions with OR (only one needs to be true) or AND (both must be true). AND has precedence over OR.

```
If major = "CS" OR major = "ACCT" AND class = "Sr" then
    outResults.AppendText(lastName & vbNewLine)
End If
```

The statement above would print the name of persons who are either senior accounting majors or cs majors in any class. To get seniors with majors in either acct or cs you need to be aware of the precedence rules and use parentheses to get the correct condition as is done in the following line;

If (major = "CS" OR major = "ACCT") AND class = "Senior"

*If Then blocks:* If your whole condition and action do not fit on one line or if you have more than one action to do on a given condition, then use:

```
If condition Then
    Action1
    Action2
End If
```

*If Then Else*: If you have actions to do when the condition is both true and false

```
If condition Then
    Action 1
Else
    Action 2
End If
```

*If Then ElseIf*: If you have more than two options, using ElseIf can be more efficient than multiple If Then statements. Only one End If will be required.

```
If condition1 Then
    Action 1
ElseIf condition2
    Action 2
ElseIf condition3
    Action3
ElseIf condition4
    Action4
Else
    Default to Action5 if none of the others are true
End If
```

*Nested blocks*: If Then Elses can be nested wherever an action is called for:

```
If condition1 Then
    If condition2 Then
        Action 1
    Else
        Action2
    End If
Else
    If condition3 Then
        Action 3
    End If
End If
```

## Control Structures: Loops

*Do While:* use when you do not know the exact number of times to repeat a process

```
aNumber = InputBox("Enter a piece of data, end with -999")
Do While aNumber <> -999
    sum = sum + aNumber
    aNumber = InputBox("Enter a piece of data, end with -999")
Loop
```

*For Next:* use when you know exactly how many times you want to repeat a process  
‘This loop will calculate the sum of ten numbers entered by the user.

```
For I = 0 to 9
    aNumber = InputBox("Enter a piece of data:")
    sum = sum + aNumber
Next I
```

## Arrays

An array is a variable that holds more than one piece of data. It is like a list of data with each item in the list being associated with its position in the list. You would refer to the third item in an array called myList using myList(2). The number (or variable) in the parentheses is the position within the list. Here we are assuming the first index used is zero.

*To declare an array you must indicate the highest value index for the array:*

Here is a declaration of an array that can hold 121 integers, (positions 0 to 120)

```
Dim myList(120) As Integer
```

It is convenient and efficient to use a loop to fill an array, process an array, or print its contents.

The following loop calculates the sum of the first twelve elements in an array:

```
For position = 0 to 11
    sum = sum + myList(position)
Next position
```

*Filling two arrays from a data file:*

```
'Declare arrays and a variable for counting the number of elements in the arrays
Dim runner(75) As String
Dim time(75) As Single
Dim numElements As Integer = 0
'note that numElements was initialized to zero in the Dim statement above
'it will be used to count the array items and also as the position within the array

'Prepare the file to be read, (open it)
FileOpen(1, "Runners.txt", OpenMode.Input)

Do While Not EOF(1)
    'Read the next two data items, (a name and a time), from the file into the arrays
    'the numElements variable holds the position number, (array index), for the items being read
    Input(1, runner(numElements))
    Input(1, time(numElements))

    'increment numElements each time through the loop
    'to move to the next position in the array
    numElements = numElements + 1
Loop

' Close before reopening in another mode.
FileClose(1)
```

## Two-dimensional Arrays

Arrays can also be declared with more than one dimension. A two-dimensional array holds a table, rather than just a list. To dimension such an array, you first give the dimensions for the number of rows and then the number of columns.

```
Private Sub btnTable1_Click()
    Dim myTable(5, 3) As Integer
    Dim Row As Integer, Column As Integer
    For Row = 0 to 4
        For Column = 0 to 2
            myTable(Row, Column) = (Row+1) * (Column+1)
            'prints one row on a single line
            outResults.AppendText(myTable(Row, Column) & "    ")
        Next Column
        'moves cursor down one line to start the next row
        outResults.AppendText( vbNewLine)
    Next Row
End Sub
```

The following table is the output of the program above that uses a two dimensional array:

1	2	3
2	4	6
3	6	9
4	8	12
5	10	15

## Input

*From Textboxes:* use when you have only a few strings or values to input

```
age = txtAge.Text
```

```
firstName = txtName.Text
```

```
length = Val(txtLength.Text)
```

Val is a function that converts strings into numbers. It is not needed when the variables have been declared.

*From Inputboxes:* use when you have information that must be input by the user.

```
firstName = InputBox ("Enter a Name")
```

*From a Data File:* this allows data to be previously entered into a file from a program or using Notepad. The file will be saved with a file name.

On the next page is a sample program that uses the data file named “QuizScores.txt” and introduces the statements needed to read data from a file. The purpose of the program is to calculate and print the average quiz score for every student in the file.

```
Private Sub btnQuizAverage_Click(sender As System.Object, e As System.EventArgs) Handles
    'this program will calculate and print the average quiz score for each student
    'the information needed is input from a data file
    Dim avg As Single = 0
    Dim student As String = ""
    Dim q1 As Integer = 0, q2 As Integer = 0, q3 As Integer = 0

    'file must be in bin/Debug folder for this project
    FileOpen(1, "QuizScores.txt", OpenMode.Input)

    Do While Not EOF(1)
        'get input from the file for one student
        Input(1, student)
        Input(1, q1)
        Input(1, q2)
        Input(1, q3)
        avg = (q1 + q2 + q3) / 3
        outResults.AppendText(student & " earned an average of " & avg &
            vbNewLine)
    Loop
    FileClose(1) 'close the file on channel 1
End Sub
```

## Output

### *Into RichTextBoxes:*

Use the name of your RichTextBox with .AppendText( ) added to it which appends whatever text you put inside of the parentheses to the text already in the RichTextBox.

Things in quotation marks get printed exactly as written.

Variables can be concatenated to the text message using an ampersand, &, and have their contents printed.

A line like outResults.AppendText( vbCrLf) moves the cursor to the beginning of the next line.

*Format Functions:* these functions format numbers and dates into the usual forms. The samples below illustrate how they might be used and the results that would be produced.

outResults.AppendText( FormatNumber (1234.5678, 3) ) → 1,234.568

outResults.AppendText(FormatCurrency(1234.5678, 2)) → \$1,234.57

outResults.AppendText(FormatPercent(.654, 2)) → 65.40%

### *Message Boxes*

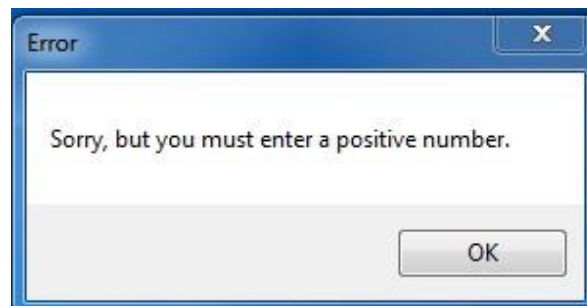
Another way to display output, besides the rich text box, is to use a Message Box. Like InputBoxes, MessageBoxes pop onto the screen and are there only until you click OK close them. Message boxes are used to display a message and then close when you click OK. This is particularly useful for error messages. A statement of the form

MessageBox.Show(prompt, title)

pops up a box with the title on the top and the prompt inside. Both the prompt and the title should be Strings in quotation marks. The statement

```
MessageBox.Show("Sorry, but you must enter a positive number.", "Error")
```

would produce the following output:





The following example of a MessageBox incorporates the value of the variable, *correctAnswer*, into the message.

```
MessageBox.Show("Sorry, the answer was " & correctAnswer & ", try again.", "Wrong Answer")
```

## Multiple Forms

To switch between forms you can have a button on your form that contains the code to hide the one you are on and show the other one. When you want to refer to the form you are currently on with your VB program, you cannot refer to it by name, but must refer to it as Me. Sample code for a button to switch control from one form to a second form named frmSortData is shown below:

```
Private Sub btnSwitchForms_Click(sender As System.Object, e As System.EventArgs) Handles btnSwitchForms.Click
    Me.Hide()
    frmSortData.show()
End Sub
```

## Sample Visual Basic Programs

Listed below are two sample VB programs including sample algorithms as comments within the code:

-----

```
Option Explicit On
Public Class frmTemperatureConversion
    'This program accepts a Celsius temperature value input by the user
    ' and then calculates and displays the corresponding Fahrenheit temperature.

    'Sample algorithm to accomplish this task:
    '1. get a Celsius temperature from the user, call it celsiusTemperature
    '2. Convert celsiusTemperature to the Fahrenheit scale and call the result fahrenheitTemperature
    ' (Use the formula:  $F = (9/5)C + 32$ )
    '3. Display the results in a complete statement including both temperatures.

    Private Sub btnConvert_Click(sender As System.Object, e As System.EventArgs) Handles btnConvert.Click
        'this is a program to convert Celsius temperatures to Fahrenheit
        'declare the variables
        Dim celsiusTemperature As Single = 0.0, fahrenheitTemperature As Single = 0.0

        'assign the user-input from the textbox to a variable
        celsiusTemperature = txtCelsius.Text

        'perform the calculation and assign the result to a variable
        fahrenheitTemperature = 9 / 5 * celsiusTemperature + 32

        'clear the rich text box before displaying result
        outResults.Clear()

        'display the result in a textbox on the form
        outResults.Text = celsiusTemperature & " degrees Celsius is equal to "
        outResults.AppendText(FormatNumber(fahrenheitTemperature, 3) & " degrees Fahrenheit.")
    End Sub

    Private Sub btnQuit_Click(sender As System.Object, e As System.EventArgs) Handles btnQuit.Click
        'stop the run of the program
    End Sub
End Sub
End Class
```

\*\*\*\*\*

Option Explicit On

**'This program will find and print the sum of the integers 'from one to N, where N is entered in a textbox by the user.**

**'this program also displays the numbers from one to n, and the running 'total while finding the sum.**

'Sample algorithm to accomplish this task:

'1. Get the value of N from the user.

'2. Initialize a variable, Sum, to the value zero.

'3. Initialize a variable, J, to the value one.

'4. Add the value of J to Sum

'5. Display the value of J and Sum

'6. Add one to the value of J

'7. If J is less than or equal to N, then repeat steps 4 through 7.

'8. Display the value of Sum

'(use a complete statement such as: "The sum of the numbers from 1 to 5 is 15")

Private Sub btnFindSum\_Click()

    'Declare variables

    Dim n As Long = 0, j As Long = 0, runningTotal As Long = 0

    Dim frmtStr As String = "{0,5}{1,25}"

    'input ending value into a variable

    n = txtGetN.Text

    runningTotal = 0

    'Print header for the table

    outResults.AppendText(String.Format(frmtStr, "J", "Running Total") & vbNewLine)

    outResults.AppendText("\_\_\_\_\_ " & vbNewLine)

    'for loop to add numbers from one to N.

    For j = 1 To n

        'add the current value of J to the running total

        runningTotal = runningTotal + j

        outResults.AppendText(String.Format(frmtStr, j, runningTotal) & vbNewLine)

    Next j

    'Print the final sum

    outResults.AppendText("\_\_\_\_\_ " & vbNewLine)

    outResults.AppendText("The sum of the numbers from 1 to " & n & " is " & runningTotal)

End Sub

-----

\*\*\*\*\*begin program about runners\*\*\*\*\*

Option Explicit On

Public Class frmRunnerForm

---

'The first button/program in this project reads a file of names and yards-gained into arrays,  
' calculates and displays the total yards gained and the average yards per runner.  
'The second button/program prints out a list of runners and their yards for those  
'runners who gained more than the average number of yards.

'Declare form level variables (needed by more than one subroutine)

Dim yardsGained(50) As Single, runner(50) As String

Dim numElements As Integer = 0, runningtotal As Integer = 0, average As Single = 0.0

Dim frmtStr As String = "{0,-20}{1,8}"

---

Private Sub btnReadFileAndTotal\_Click(sender As System.Object, e As System.EventArgs) Handles

numElements = 0 'initialize to zero, to be used for position in the array

'Prepare the file to be read

FileOpen(1, "Runners.txt", OpenMode.Input)

'print the header info

outResults.AppendText(String.Format(frmtStr, "Runner", "Yards Gained") & vbNewLine)

outResults.AppendText("\*\*\*\*\*" & vbNewLine)

Do While Not EOF(1)

'Read next data set from the file into the array and print the data

Input(1, runner(numElements))

Input(1, yardsGained(numElements))

' note comma is used as a line continuation character

' to continue long line of code over two lines.

outResults.AppendText(String.Format(frmtStr, runner(numElements),  
yardsGained(numElements)) & vbNewLine)

'add yards to runningTotal

runningtotal = runningtotal + yardsGained(numElements)

'increment each time through the loop

'to move to the next position in the array

numElements = numElements + 1

Loop

FileClose(1) 'Close the file used for input

'print the net yards gained by all runners

outResults.AppendText("\*\*\*\*\*" & vbNewLine)

outResults.AppendText("Net yards gained = " & runningtotal & vbNewLine)

average = runningtotal / numElements

outResults.AppendText("The average yards gained = " & average & vbNewLine)

'disable the command button for Reading the file

btnReadFileAndTotal.Enabled = False

'enable the command button for printing the above average runners

btnAboveAvg.Enabled = True

End Sub



```

Private Sub btnAboveAvg_Click(sender As System.Object, e As System.EventArgs) Handles btnAboveAvg.Click

    'print the header info
    outResults.AppendText("*** Runners Who Ran For More Than The Average Number of Yards ***" & vbNewLine)
    outResults.AppendText(String.Format(frmtStr, "Runner", "Yards Gained") & vbNewLine)
    outResults.AppendText("*****" & vbNewLine)

    For position = 0 To numElements - 1
        If yardsGained(position) > average Then
            outResults.AppendText(String.Format(frmtStr, runner(position),
                                                yardsGained(position)) & vbNewLine)
        End If
    Next position
End Sub
End Class
*****end program about runners*****

```

\*\*\*\*\* Connecting to a Database \*\*\*\*\*

VB can be used to connect to an Access database. The method we are choosing to demonstrate relies on a DataGridView object being added to a VB project and bound to an Access database table or query. When you attempt to add a DataGridView object to a VB form, there are many menus that will pop up and that you need to interact with in order to tell the program what kind of connection you would like to make and what data source you want to connect to, typically a table or query from a database. Once that data grid has been assigned a collection of records through that binding process, you can write VB code to interact with the data. Your code will interact with the data by referencing it through the DataGridView. The DataGridView makes it easy to reference an item from a table by referring to it in terms of row and column offsets, similar to the concept of position within an array and essentially identical in concept to referring to a location in a two-dimensional array. (both the row and column location are needed)

Below is a sample VB program that connects to the BigFamily table of an Access database called Family.accdb and lists the name and birthdate of those family members born in the month of August. You can assume that the DataGridView referred to in the program, *dgvBigFamily*, was first bound to the table with information about the family members when it was added to the VB form for this program.

```

Private Sub btnAugustBirthdays_Click(sender As System.Object, e As System.EventArgs) Handles btnAugust
    Dim row As Integer = 0, birthday As Date = #8/7/2012#
    Dim searchMonth As Integer = 8 ' eight is for August

    outResults.AppendText("August Birthdays" & vbNewLine)

    For row = 0 To dgvBigFamily.RowCount - 1
        birthday = dgvBigFamily.Rows(row).Cells(3).Value

        ' The Month function will return the integer representing the month (for August, that is 8)
        If Month(birthday) = searchMonth Then
            outResults.AppendText(dgvBigFamily.Rows(row).Cells(0).Value &
                " " & dgvBigFamily.Rows(row).Cells(3).Value & vbNewLine)
        End If
    Next row
End Sub

```



# ***Appendix B Excel VBA Macro Reference Guide***

## **Recording A Macro**

- 1) Position cursor
- 2) Choose Developer - Code - Record New Macro
- 3) Name macro and give it a control character
- 4) You are now recording. Choose relative addressing if appropriate.

Relative addressing will use addresses that will change automatically if you run this macro with the cursor in a different location than at the present time. All cell references will be in relation to where the current ActiveCell is when the macro is run.. If you want your macro to always refer to specific cells, use absolute addressing which means the macro will always use exactly those cells that you refer to in your macro.

Every keystroke you enter at this time will be recorded. Do whatever task you had in mind (ie. Enter formulas, text, and formatting to accomplish the task.)

- 5) Stop recording
- 6) Edit macro using Developer – Code - Macros - Edit.

## **Writing your own VB Macro**

- 1) Record a new macro, recording only the name, control shortcut, and then stop the recording.
- 2) Open the macro for editing and enter your VB text
- 3) Test and Edit macro

## **Input and Output from a Spreadsheet**

1. To place program output in a spreadsheet cell:

```
Range("Result") = Sum
ActiveCell = hours * payrate
ActiveCell.Offset(3,0) = total
ActiveSheet.Cells (11,9) = Sum
Sheet2.Cells(Row, 1) = ActiveCell
```

2. To input from a spreadsheet cell into a variable:

```
Amount = ActiveCell
Amount = ActiveCell.Offset(0,3)
Amount = ActiveSheet.Cells(1,5)
```

3. *Input Boxes*: Assign the results obtained from an Input Box statement to an active cell or an offset cell. The input will go into the assigned cell.

```
ActiveCell = InputBox ("Enter a number:")
ActiveCell.Offset(0,4) = InputBox("Enter the name of the item")
```

To move about on a Spreadsheet

1. To move to a specific cell and make that cell the Activecell:

Range("B3").Select

Or

Range("myStartingPoint").Select    'where myStartingPoint is a named range

2. To reference a cell other than the current ActiveCell, use Offset:

If ActiveCell.Offset(0,-1) = "Finance" Then

    ActiveCell = ActiveCell + 1000

End If

(checks cell one column left of the current cell for "Finance" and adds 1000 to the current cell)

3. To move down from row to row (within a loop):

ActiveCell.Offset(1,0).Select

When using the Offset method: (row, column)

Positive numbers refer to cells down or to the right,

Negative numbers refer to cells up above or to the left of the current cell.

## Loops

Two formats that are used often are demonstrated below, Do While and For/Next.

Do While ActiveCell.Offset(Row,0) <> ""    'process cells until an empty cell is reached

    'put your list of actions here

    Row = Row + 1

Loop

For Row = 5 to 30                    'process a certain number of cells

    'List of actions might include a reference to ActiveCell.Offset (Row, 2) where

    'Row refers to cells that are offset from the active cell by from 5 to 30 rows

    sum = sum + ActiveCell.Offset (Row, 2)

Next Row



## Sample Excel VBA Macros

Listed below are two versions of the two different macros. In the first version of each, the cursor remains stationary once it is moved to the starting point. All cell references are made using offsets from the starting point. In the second version the cursor is moved throughout the macro and the cell reference used is the Activecell. The task is accomplished in both cases. In this course, CSCI 130, we generally prefer that the cursor remains stationary when processing a row or column of cells and that the offsets are used to reference the desired cells.

```
*****
Option Explicit

Sub a_1_HowManyOrdersOver500_Items()
' Note that the cursor will stay in one place
' 1. HowManyOrdersOver500_Items Macro
'
' Keyboard Shortcut: Ctrl+o
'
' Ctr will be used to keep track of how many orders have quantities over 500.
Dim CTR As Integer, rowOffset As Integer
rowOffset = 0

'move to starting point
' "firstQty" is a defined Range name on the spreadsheet, (cell C3)
Range("firstQty").Select

'look through the orders and count the ones having quantities over 500
Do While ActiveCell.Offset(rowOffset, 0) <> ""
    'check the QTY
    If ActiveCell.Offset(rowOffset, 0) > 500 Then
        CTR = CTR + 1
    End If

    'you can use the Debug.Print statement to assist you in finding errors, when needed
    ' the Debug.Print statements print to the Immediate window at the bottom of the screen
    'To make the Immediate window visible, from the menu, choose View, Immediate Window
    Debug.Print "the value of rowOffset is"; rowOffset
    rowOffset = rowOffset + 1 'refer to the next row the next time through the loop(the next Qty)
Loop

'Display results
MsgBox ("There were " & CTR & " orders with more than 500 items in each.")

'place cursor in corner when finished
Range("A1").Select

End Sub
*****
```

```

Sub a_1b_HowManyOrdersOver500_ItemsVersion2_w_cursor_movement()
' Note that the cursor will move from cell to cell
' 1. HowManyOrdersOver500_ItemsVersion2_w_cursor_movement Macro
'
'Ctr will be used to keep track of how many orders have quantities over 500.
Dim CTR As Integer

'move to starting point
' "firstQty" is a defined Range name on the spreadsheet, (cell C3)
Range("firstQty").Select

'look through the orders and count the ones having quantities over 500
Do While ActiveCell <> ""
    'check the QTY
    If ActiveCell > 500 Then
        CTR = CTR + 1
    End If

    'move the cursor to the next cell down in the column
    ActiveCell.Offset(1, 0).Select
Loop

'Display results
MsgBox ("There were " & CTR & " orders with more than 500 items in each.")

'place cursor in corner when finished
Range("A1").Select

End Sub

```

---

\*\*\*\*\*

## Option Explicit

‘Program to search down column B for all people in the Finance department and to list their ‘names (from column A) on Sheet2, starting in the seventh row:

```

Sub FinanceList()
    ‘FinanceList_Version_1_stationary_cursor Macro
    ‘keyboard shortcut: Ctrl + f
    Dim place as Integer, row as Integer
    place = 7
    row = 0
    Range(“B2”).Select    ‘positions cursor in cell B2 on the current sheet, (sheet1)

    ‘the following loop repeats as long as the cell referred to is not empty
    Do While ActiveCell.Offset(row,0) <> ""
        If ActiveCell.Offset(row,0) = "Finance" Then
            Sheet2.Cells(place,1) = ActiveCell.Offset(row, -1)
            place = place + 1    ‘refer to next cell on sheet2 for the next entry
        End If
        row = row + 1    ‘refer to the next cell in the department column
    Loop
End Sub

```

\*\*\*\*\*

### Excel VBA Macro to Read from a File:

Data can also be read from a data file onto a worksheet by opening the file and using a loop to assign the data to a cell and then changing which cell is being referred to by incrementing the row or column offset variable. At the beginning of the next page is a sample VBA macro that reads data from a file and puts it in a spreadsheet:

```
Sub ReadDataFile()  
'  
' ReadDataFile Macro  
' This macro will read a file containing snowfall data  
' and place the data in the spreadsheet.  
' Each line in the file contains two items, a city name and snow depth.  
  
    'declare variables  
    Dim row As Integer, city As String, snow As Single  
  
    'initialize variables  
    row = 0  
  
    'open data file to be read  
    Open ActiveWorkbook.Path & "\snowdepth.txt" For Input As #1  
  
    'place cursor in desired position  
    Range("A2").Select  
  
    'read the data file  
    Do While Not EOF(1)  
        'get the data from the file  
        Input #1, snow, city  
  
        'put data in the spreadsheet  
        ActiveCell.Offset(row, 0) = city  
        ActiveCell.Offset(row, 1) = snow  
  
        'increment row counter to refer to next row  
        row = row + 1  
    Loop  
  
    'close the data file before exiting the program  
    Close (1)  
End Sub
```

### Assigning a VBA macro to a Button on a Spreadsheet:

Write the macro using the VB editor and then return to the spreadsheet. Use the menu choices to put a button on your spreadsheet and assign a VBA macro to it. On the Developer ribbon, in the *Controls* group, click on Insert, and then on the Button icon. Move over to the location you want for the button and drag the mouse to create the button. Another menu will pop up and allow you to assign the prewritten macro to this button and also allow you to change the caption on the button.

### **Other Tips**

When editing code, both your VB page and your spreadsheet will appear in your bottom toolbar.

To use a worksheet formula in a VB program use `Application.WorksheetFunction.name of function`. Or record the function and then you can see what the syntax should be or then just copy the name into your program.

Eg. `ActiveCell = Application.WorksheetFunction.Average(Exam1, Exam2, Exam3)`

# ***Appendix C Access VBA Macro Reference Guide***

## **Creating your own VBA script (VBA Macro)**

Assuming you have designed a solution to a task you would like to implement, bring up a form for a database in design view. De-activate the “Use Controls Wizards” icon in the Controls group by clicking it once so they do not activate when adding a command button to your form. Add a command button to your form. Once the button is on your form, right-click on the button to bring up a popup menu of options and click on Properties to see attributes related to the button. If you click the ALL tab, a complete list of the attributes is displayed. Change the Name and Caption attributes to something appropriate for the task you are associating with the command button, (for example, btnCalculateAverageManagerSalary and “Calculate Avg Mgr Pay”)

Scroll down the attribute list until you come to “On Click”. When you click in the blank space to the right of “On Click”, two icons will appear, a down arrow and three dots. Click on the three dots to bring up another menu with three choices, Expression Builder, Macro Builder, and Code Builder. Choose Code Builder to take you to a Visual Basic development environment where you can write the VBA code to accomplish the desired task. The beginning and ending lines of your program, (Private Sub btnCalculateAverageManagerSalary\_Click() and End Sub) are written for you already. Type the VBA code for your solution between those two lines. All of the code between those two lines will be executed each time the command button is clicked.

A summary of these steps is listed below for easy reference:

## **Writing your own VBA script**

- 1) Add a button to your form and give it an appropriate name and caption.
- 2) Right-click the button
- 3) Choose Build Event
- 4) Choose Code Builder
- 5) Type in your code
- 6) Test and edit

## **Variable declaration**

Variables are declared using a Dim statement that includes the name of the variable and the datatype to be used. Although all variables do not have to be declared, it is good programming practice to declare the variables used to assist in documentation and debugging.

```
Dim maxSalary As Single
```

```
Dim firstName As String
```

When writing VBA scripts for Access 2007, you usually work with a collection of records that are part of some database. VBA has several special datatypes to facilitate working within a database application. The datatypes Database and Recordset are used to declare object variables that can then be assigned values and manipulated using the many methods available for each object type. To declare a variable of one of these special datatypes, use the Dim statement just like any other variables in Visual Basic.

```
Dim myDatabase As Database
```

```
Dim myRecords As Recordset
```

## Variable Initialization

Simple assignment statements can be used to give values to most variables as is done in Visual Basic.

```
maxSalary = 72500
```

```
firstName = "Hal"
```

When variables of the "special" datatypes created for Access are used,(ie. Database or Recordset), the initialization or assignment statement is prefaced by the word **Set**.

```
Set myDb = CurrentDb()
```

CurrentDb() is a function that returns a copy of the database currently being used

Another way to initialize a database variable using the complete path to the database.

```
Set myDb = OpenDatabase( "M:\CS130\AccessSamples\Gargoyles.mdb")
```

```
Set myRecords = myDb.OpenRecordset("EmployeesTable")
```

"EmployeesTable" is the name of an existing table in myDb

## Constants

The programmer can declare the equivalent of read-only variables which can be given values that will not change throughout the run of the program. The declaration must be prefaced with Const.

```
Const Pi as Long = 3.14159
```

VBA has several named action constants that are not pre-defined in terms of their value, but in terms of a relative location. The action constant acNext refers to the next record in a recordset. Action constants like acNext are used as parameters in methods such as GoToRecord.

```
DoCmd.GoToRecord , , acNext
```

## Methods

Methods are the means of manipulating objects and or communicating among objects in an object-oriented programming environment. For example, in Visual Basic the Rich TextBox object has an AppendText method that allows the programmer to display information in a Rich TextBox. The syntax is generally the name of the object followed by the name of the method.

```
outResults.AppendText("This message will be printed in the rich text box.")
```

In VBA for Access there are two special purpose objects called Application and DoCmd that have many associated methods for accomplishing some of the most commonly required tasks in working with a database. The Application object refers to the Access program and is the default object if none is given. This allows the user to shorten some statements,(ie Application.Quit can be written as simply Quit.) The DoCmd object is used to accomplish many of the same actions used in building macros. Some of the methods available for DoCmd are:Beep, GoToRecord, OpenForm, Open Report, OpenQuery, OpenTable, Quit, and ShowAllRecords. When you are typing in your VBA code a pop-up menu listing all of the methods will appear once you type DoCmd and a period. Some of the methods require one or more parameters that must be put in a designated order. If a parameter is optional, you may need to use a comma as a placeholder to indicate the correct position for the parameter. The two statements below are equivalent, one having omitted the optional parameters.

```
DoCmd.GoToRecord , , acNext
```

```
DoCmd.GoToRecord acDataTable, "Employees", acNext
```

## Recordsets

There are five different types of Recordset objects used in VBA. They are Table- type, Dynasettype, Snapshot-type, Forward-only-type and Dynamic-type. The type of Recordset used in an application depends on the source of the records and the desired features for updating the underlying records. Access will automatically choose the best-performing Recordset type as the default if the programmer does not specify which one to use. You must specify the type of Recordset in the initialization statement if it is other than the default Dynaset-type allows the source of the records to be a query and also reflects the changes in the Recordset made to the underlying records by other users, (when the database allows for multiple users), while Tabletype Recordset objects do not allow such updates. There are many other issues concerning the choice of Recordset type that are beyond the scope of this introduction to VBA.

### *Indexing a recordset object*

After declaring a recordset object using the Dim statement and initializing it using the Set statement, the order the records are accessed can be set using the index method. The index method allows the programmer to control the order in which the records are traversed using the methods for moving around a recordset listed in the next section.

Here is an example of declaring, initializing, and setting the index of a recordset object before using it.

```
Dim myWorkers as database
Dim myEmployees As Recordset
Set myWorkers = CurrentDb()
Set myEmployees = myWorkers.openRecordset("Employees")
myEmployees.Index = "EmployeeID"
(Where EmployeeID is the name of an existing index for the Employees table)
```

### *Traversing the recordset*

There are several methods that can be used with a Recordset object to move around the recordset. Some of those methods are: MoveNext, MovePrevious, MoveFirst and MoveLast

These methods would be used in conjunction with a Recordset object to move to a particular record in a collection of records.

MyRecords.MoveNext      (move to the next record)

MyRecords.MoveLast      (move to the last record)

### *Accessing Field Values of the Current Record*

The syntax for referring to the value of a field within the current record of a Recordset object uses an exclamation point.

If myRecords!Salary = 15000 then MsgBox(myRecords!FirstName & " is underpaid.")

The line above uses a Recordset object named myRecords having FirstName and Salary fields.

When using the default recordset, you can omit the Recordset name. The record referred to is the current record being displayed on the form.

If [Salary] = 15000 then MsbBox([FirstName] & " is underpaid.")

### *Making Changes to a Record Within a Recordset*

To make changes to a record in a recordset object, the record first needs to be copied to a copy buffer and after the changes are made the copy buffer needs to be saved. VBA has two methods to do this. The Edit method copies the current record to the copy buffer to allow it to be edited and the Update method saves any changes made to the record while it was in the copy buffer.

As an example if you have a recordset object called myEmployees and you'd like to change the FirstName of the current record from Harold to Hal, the following lines of code would accomplish that task.

MyEmployees.Edit

MyEmployees!FirstName = "Hal"

MyEmployees.Update

### *Referring to the Current Record*

The reserved word, "Me", is used in VBA for to refer to the current record. "Me" acts as the object name for the current record and allows the programmer to use object methods that act on the current record. For example, using the VBA statement , "If Me.newRecord then Quit", translates to "If the current record is an empty new record, then quit the application".



### *End of Recordset Indicator*

VBA has a boolean method called EOF that is used with recordset objects. A call to EOF returns true if the end of the recordset has been reached and returns false otherwise. The EOF method is often used as part of a loop control expression.

Do While (Not myRecords.EOF)

There is a similar method, BOF, that refers to the beginning of the recordset.

### **Debugging**

VBA has a Debug object that is convenient to use in conjunction with the associated Print method to assist the programmer in developing error-free code. The Debug.Print statement prints the results in the "Immediate" window of the VBA environment. Inserting temporary print statements at various points in your code can help you ensure that your code is doing what you want it to do.

Debug.Print (" The current employee ID is " & myEmployees!ID)

### **VBA Macros: An Example**

Built-in macros are useful for many common tasks, such as navigating from record to record, adding or deleting a record, or sorting records into a desired order. However, often a user would like to do tasks for which there are no built-in macros available. The user can design and write his/her own VBA code within the Access database to accomplish a desired task and then assign that code to be executed on the “On Click” event of a button on a form or one of many other events.

We will be using just a limited subset of VBA’s capabilities to introduce you to working with VBA and Access. This will give you the tools to go beyond the limited logic available in the built-in macros and to learn to design and code solutions specific to a particular database. Just as in writing a Visual Basic program, before you attempt to use VBA to accomplish a task you must decide exactly what you wish to accomplish and design an algorithm to get the job done. Having a plan to solve the problem is over half of the challenge. A well thought out algorithm, whether written in pseudo code or shorthand, will be of great benefit in getting to a working solution more quickly.

Here is a sample VBA program that is designed to look through the records of the “Employees” table of the “Gargoyle” database and find the record of the highest paid employee. Once all of the records have been searched, the record with the highest salary is displayed and a message box pops up displaying the employee’s name and salary. The macro is assigned to a button on the “Employees” form. When the button is clicked the code below is executed. The program is referred to as an event procedure since it is the process that is followed in the event of a mouse click on the button.

```

Private Sub cmdMaxSalary_Click()
    'This code segment locates the record with the Maximum Salary
    'A declared Recordset object is used in this script
    'A message box displays the amount and the person's name.
    'The Employees form display also moves to the record with the Maximum salary

    'declare variables
    Dim myDb As Database
    Dim myRecords As Recordset
    Dim maxSalary As Single
    Dim recordN As Integer, ctr As Integer
    Dim highestPaidFirst As String, highestPaidLast As String

    'initialize variables
    Set myDb = CurrentDb()
    Set myRecords = myDb.OpenRecordset("Employees")
    myRecords.Index = "PrimaryKey"
    maxSalary = 0
    ctr = 0

    'position on desired record before beginning task
    'Not needed here since opening a Recordset moves to first record

    'Loop to search for the highest salary, beginning at the first record
    Do While Not myRecords.EOF
        ctr = ctr + 1    'counts each record
        'sample debugging statement to see the order the records are accessed
        'Debug.Print myRecords!FirstName
        If myRecords!Salary > maxSalary Then
            maxSalary = myRecords!Salary
            highestPaidFirst = myRecords!FirstName
            highestPaidLast = myRecords!LastName
            recordN = ctr 'remembers which record is the Max so far
        End If
        myRecords.MoveNext    'Move to the next record in the set
    Loop

    'position display at the highest paid employee's record
    DoCmd.GoToRecord, , acGoTo, recordN

    'Print a formatted message summarizing desired results
    MsgBox (highestPaidFirst & " " & highestPaidLast & " has the max"
        salary ==> " " & FormatCurrency(maxSalary))

End Sub

```

Simple computer programs typically can be broken down into three parts:

1. Get ready to do the desired task by declaring and initializing variables and positioning the cursor at the desired starting point.
2. Do the desired task, such as searching for records or performing calculations.
3. Display the results, if desired.

The prior VBA program has all three of the typical parts.

The statements beginning with Dim are declaration statements making several variables available to be used as part of the program. Immediately after the Dim statements are two lines that assign the current database and the records from a particular table to the values of the database and Recordset variables. Note the use of the “Set” command as part of this assignment statement. “Set” is necessary here, but note that it is not used in most assignment statements. The next line assigns an order in which the records will be accessed. The next two lines assign the value of zero to two variables.

The DoCmd object has several methods that assist the programmer in completing common tasks such as record navigation. The methods that are used with the DoCmd object are very similar in function to the “actions” used in building macros. The commas in this line are place holders to ensure that the constant, acFirst, is interpreted to be the third argument for the method. The other arguments in this case are optional.

The main task, in this case, is accomplished with the use of a Do While loop to repeat the activity of checking for the highest salary with every record in the table. The body of the loop, (everything between Do While and Loop), contains an IF statement to ask if the current record’s salary field is the biggest salary encountered so far and also a statement to move on to the next record.

After the loop there are two lines of code; one to position the display to the desired record and the second to print the results. Note that lines beginning with a single quote are comments, not executed when the program is run.

### *Declared Recordset Objects or Default Recordset Objects*

VBA does not force the programmer to always declare recordset objects. In some situations, the task you want to accomplish may involve only the records from a single table that are associated with a particular form. If that is the case, when the form is created a default recordset object is also created, allowing for a simpler syntax in referring to the fields of the current record. When a default recordset is used, the programmer does not need to use a Recordset object name in referring to a field of the current record, but can just use the field name by itself enclosed in square brackets. (ie. [FirstName] instead of myRecords!FirstName)

Here is another version of the previous sample program, rewritten to use the default record set.

```
Private Sub btnMaximumSalaryUsingDefaultRecordset_Click()  
    'This code segment locates the record with the Maximum Salary  
    'The default Recordset associated with the form is used.  
    ' A message box displays the value and the person's name.  
    ' The Employees form display also moves to the record with the Maximum salary  
  
    'declare variables  
        Dim maxSalary As Single  
        Dim HighestPaidFirst As String, HighestPaidLast As String  
        Dim recordN As Integer, ctr As Integer  
    'initialize variables  
        maxSalary = 0  
        ctr = 0  
    'position on desired record before beginning task  
        DoCmd.GoToRecord , , acFirst  
  
    'Loop to search for the highest salary, beginning at the first record  
    Do While (Not Me.NewRecord)  
        ctr = ctr + 1  
        'Debug.Print [FirstName]  
        'sample debugging statement to see the order of records accessed  
        If [Salary] > maxSalary Then  
            maxSalary = [Salary]  
            HighestPaidFirst = [FirstName]  
            HighestPaidLast = [LastName]  
            recordN = ctr  
        End If  
        'Move to the next record in the table  
        DoCmd.GoToRecord , , acNext  
    Loop  
  
    'position display at the highest paid employee's record  
    DoCmd.GoToRecord , , acGoTo, recordN  
  
    'Print a formatted message summarizing desired results  
    MsgBox (HighestPaidFirst & " " & HighestPaidLast & _  
        " has the max salary ==> " & ormatCurrency(maxSalary))  
End Sub
```