# CS 105 – Introduction to Computer Science

The purpose of this handout is to guide you through the first unit of the course, which deals with computer problem solving. This is your reading material to prepare for class. As with any course you take, don't fall into the trap of doing nothing during the early weeks of the course. The material will accumulate, and you need to apply your best study habits right now from the beginning. Don't wait. A month from now … you will be glad (or you will have wished) you started today.

## Preliminary Remarks

In essence, computer science (CS) is the study of how to solve problems. Also, we are interested in how to represent various forms of information. Computer scientists study various ways to solve problems and the nature of problems themselves. We classify problems, and seek ways to solve them systematically. As you might expect, the types of problems addressed in CS are usually quantitative in nature. However, the information that our problems deal with are not always numbers. Information may include text, images, sounds and video.

You might have noticed that my definition of CS above did not involve the word "computer." The computer is merely a general-purpose machine that assists us in solving problems much more rapidly and reliably than just working manually. So, this course is about how to think, and how to solve problems, especially using a computer.

What does it mean to "solve a problem" in this course?

For example, here's a problem: "Calculate a grade point average." The solution to this problem is not just a number like 2.4. This value may or may not be my GPA or somebody else's. Instead, the way to solve a problem is to *provide the necessary steps to get to a solution*. This way, you can obtain the grade point average for anyone, not just for one person.

Throughout this course, it may help to think about cooking as an analogy. After all, every day we personally have the problem of "I am hungry." A cookbook is not just a bunch of pretty pictures of food. It also gives you the instructions you need to follow to create dishes.

This is an important point, so it bears repeating: To solve a problem, we need to write down the steps necessary to get us to an answer. When we are done, the solution to a problem is called an "algorithm."

How do we know if we are right?

If we solved a problem correctly, this means that we will always get the right answer when we follow the steps. In other words, when we work through the steps, either by ourselves or on the computer, and our final result is wrong, then this means that there is a mistake in our solution. (Of course, it's possible that we followed the steps incorrectly, but the computer won't make that kind of mistake!)

Since the solution to a problem is not simply a single word or number but rather a list of instructions, we should expect the problem-solving process to take a while. So, we need to be patient with ourselves, and have faith in our abilities.

I'd like to take this opportunity to reiterate some guiding principles for learning.

1. We are here to learn and explore.

   a. Seek discussions with the instructor and classmates about the material to reinforce your understanding and practice communicating ideas.

   b. Have fun. Live in the moment (i.e. don't dwell too much on the difficulties of yesterday or tomorrow). Enjoy the journey and intellectual feast. Be enthusiastic about what you are doing.

2. You can be successful in this class. Every day is an opportunity for an epiphany. Don't let mistakes or setbacks hold you back. After some effort, things can suddenly click in your mind.

3. Learn by doing!
   Not just passively reading, listening or watching.
   Each study period needs to have a clear goal.
   Pay attention to the big picture and the facts that you are collecting.

4. Be organized: Take notes on what you read. Review earlier material as needed. Create a cumulative study outline, and update it each week. Maintain a portfolio of your work.

5. Be patient when solving a homework or lab problem.

   a. There is no need to rush.
      Don't worry if your first attempt at a solution is wrong.
      Read all instructions and be methodical.
      Take time to gather your thoughts.
      Deliberately write out your thought process and plan of attack.

   b. A computer program or other homework assignment may take up to several hours to complete. In programs you need to comment your code as you go, because you will quickly forget what looks obvious right now! Realize that you don't need to finish everything in one sitting.

   c. Break up large problems into small, more manageable pieces.

   d. Don't get bogged down with too many mechanical details. Computing is all about removing tedium from routine tasks.

6. Be curious, and always ask questions.

   a. Find a topic or application that you are enthusiastic about.

   b. Consider alternative solutions to a problem.

c. When finishing a problem, ask yourself if this problem or its solution lends itself to other problems.

7. Computer science is about logic, structured thinking, information, communication and problem solving. Thus, it has connections to many other fields in the sciences, humanities and social sciences. You will find the analytical techniques useful in your career.

In every course you take, including this one, it is sometimes necessary to answer a "what is" question as you study. Depending on the application (e.g. cheetah, sautéing, freedom, Canada, airplane, computer science), here are some ways you can approach such a question.

- What it consists of, or its properties
- What it resembles, is analogous to
- How it behaves
- How life would be different without it
- What it is NOT
- Its purpose

In this course we will study two things:

- How to solve problems in general using the computer
- How to apply these principles to a particular problem domain

Discussion:

1. If you had to describe the computer with one word, what would it be?

2. What is the role of the computer in computer science?

3. Find a cash register receipt from a recent trip to the store. How would you describe or characterize the information contained in it? What information on it is unimportant to you? Give an example of a labeled value on the receipt. What is the label, and what is the value?

4. Since you woke up this morning, what sort of "problems" did you need to work out in your mind? Did you follow a sequence of steps that you figured out some time ago?

5. As suggested above, we can practice a "what is" type of question. For a given noun (e.g. cheetah), decide which strategies for answering a "what is" question are most appropriate.

# BINARY NUMBERS

Let's begin with some basics: how information is represented inside the computer.

*Information* includes such things as: numbers, text, sounds, images and video. All information must be represented as 0s and 1s when stored insides the computer. The word *binary* refers to a number system using only 0s and 1s. Binary is the computer's native language.

A *bit* is the smallest unit of information. It is a single 0 or a single 1. The word "bit" is short for "binary digit."

The basic building block of computer hardware, such as the Central Processing Unit (CPU), is the *logic gate*. The purpose of logic gates is to manipulate individual bit values. They can do so very quickly, in less than a billionth of a second. Various logic gates are arranged to allow the hardware to perform mathematical operations.

Since single bits are not very interesting, we tend to group them so that they can store a useful amount of information. The most common way to group bits is to arrange them in units of eight. A *byte* equals 8 bits of information. The byte is the fundamental unit of measuring amounts of memory or information. One byte is usually enough space to hold a single letter of text or a small number.

There are two essential skills that you need to learn:

- Interpreting binary: converting a binary number into decimal
- Encoding into binary: converting a decimal number into binary.

Just to clarify, the word "decimal" here means "base 10". The numbers that we will use in this course will be non-negative integers. In future courses, you will see how negative numbers and real numbers are represented in binary.

## Converting binary to decimal

How do binary numbers work?

The first thing to understand about binary numbers is that this system is base 2, rather than the base 10 (decimal) that we humans use. We write out a binary number using the same place-value system as in decimal. Let's review what the place-value system means with this example:

We understand that when we see the number 278 that this means (2 * 100) + (7 * 10) + (8 * 1). Each digit of the number 278 has to be multiplied by the appropriate power of 10. Binary works exactly the same way except for two details:

- Each digit is either a 0 or a 1; no other symbols can be used.
- Digits are multiplied by powers of 2, not powers of 10.

For example, let's look at the binary numbers 001110 and 100011:

| 32 times | 16 times | 8 times | 4 times | 2 times | 1 times | Number value |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 0 | = 14 |
| 1 | 0 | 0 | 0 | 1 | 1 | = 35 |

Notice that each digit of a binary number is multiplied by a power of 2. The rightmost digit is multiplied by 1, the digit to its left is multiplied by 2, etc. The total value of a binary number is the sum of the values we get from each digit.

At this point, it would be helpful to take a look at some powers of 2.

$2^0 = 1$
$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$

You should memorize these powers of 2, because they occur very frequently. Actually, there is very little here that needs to be memorized. At a bare minimum, you should know that $2^0 = 1$. Any of the other powers of 2 can be obtained by repeatedly doubling until you reached the desired power.

In particular, you don't need to memorize $2^5$ and $2^6$. You can figure them out just by looking at $2^4$ and doubling as needed. What are the results?

Now, since each binary digit can only be a 0 or a 1, each power of 2 is being multiplied by 0 or 1. Multiplying by 0 always gives 0. So, a 0 digit does not add anything to the value of the number. Multiplying a power of two by 1 does not change it. As a result, this process of converting from binary to decimal can be seen as just adding powers of 2. Which powers of 2? The ones where we see a "1" in the binary representation. The two examples we have above can be written as:

$2^3 + 2^2 + 2^1 = 14$
$2^5 + 2^1 + 2^0 = 35$

Therefore, if you see a binary number that begins with 0's, these can be ignored. For example, the binary numbers 1110, 01110, 00001110 and 00000000000000000000000000001110 are all equal to 14.

Sometimes it is very handy to work with much larger powers of 2. But rather than worrying about their exact values, it is often helpful to estimate them. You should memorize these approximations:

$2^{10}$ is about 1 thousand
$2^{20}$ is about 1 million
$2^{30}$ is about 1 billion

If you combine the small exact powers of two with these approximations, you can quickly come up with approximate values of many powers of 2. But you will need to use a mathematical fact that you learned in basic algebra: $a^{b+c} = (a^b)(a^c)$. For example, suppose we wanted to approximate $2^{25}$. The number 25 can be written as 25 = 5 + 20. Therefore, $2^{25} = (2^5)(2^{20})$. From our earlier work, we know that $2^5$ is 32 and $2^{20}$ is about 1 million. Thus, $2^{25}$ should be approximately 32 million.

Discussion:

1. Estimate the values of $2^{16}$ and $2^{34}$.

2. You can also go the other way: which powers of two are near the following values: 4 billion, 128 thousand, 2 million?

3. Let's say we have 4 bits. What is the lowest number we can have? What is the highest number we can have?

4. Repeat the previous question assuming we have 5 bits.

5. Is there a pattern going on here? You should be able to write a formula for the largest possible that can be written with n bits.

<u>Converting decimal to binary</u>

It has been said that, "There are 10 kinds of people in the world. Those who understand binary, and those who don't!"

One thing to note is that binary numbers are longer than decimal. It takes more binary digits to express a number than base 10 can. For example, a 5-digit decimal number may turn out to need 15 bits! In general, binary numbers are about 3 times the length of decimal numbers.

Converting decimal into binary is slightly more complicated than going the other way. Here is a technique for converting decimal numbers into binary. Let's imagine shopping that we at a very special store that I'll call the "Binary Store." All merchandise in this store is priced at a power-of-2 number of dollars: $1, $2, $4, $8, $16, $32, etc.

You enter the store with a certain number of dollars, say $45. As you shop, your goal is to *always purchase the most expensive gift possible*. And you must spend all of your money. In the case of $45, the first item we can buy is $32. We subtract in order to figure out how much money we have left: 45 – 32 = 13. Next, we find an $8 item, so that 13 – 8 = 5. With $5 remaining we see that we can buy a $4 item and a $1 item. When we check out of the store and look at our receipt, we see that we bought items costing $32, $8, $4 and $1. In other words, mathematically we have 45 = 32 + 8 + 4 + 1. Note that we did not purchase the $16 item or the $2 item. Some items got skipped based on what money we had to start with.

To write down our binary answer, we just fill in this table:

| 32 * | 16 * | 8 * | 4 * | 2 * | 1 * |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 1 |

Once we know that 45 = 32 + 8 + 4 + 1, we write down all of the powers of 2 from 32 down to 1. Underneath each power of 2, we write a 0 or a 1. Write down “1” if we bought that item, and write “0” if we did not. In this case, we bought the items costing 32, 8, 4 and 1 dollars, so those powers of 2 get 1s. But we did not buy the items costing 16 or 2 dollars, so those powers of 2 get 0s. Our final answer is to say that 45 in decimal is equivalent to 101101 in binary.

Let’s try another example. Convert 61 into binary. When we go to the binary store, we see that we can buy items costing 32, 16, 8, 4 and 1 dollars. In other words, 61 = 32 + 16 + 8 + 4 + 1. Once again, we need to write out the powers of 2, this time from 1 on the right end to 32 on the left end. Write a “1” underneath each power of 2 that we actually bought.

| 32 * | 16 * | 8 * | 4 * | 2 * | 1 * |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 1 |

Our binary answer is 111101. Incidentally, we could have written our powers of 2 using exponents, like this: $61 = 2^5 + 2^4 + 2^3 + 2^2 + 2^0$. The exponents on 2 tell us where to write the 1s in our binary answer. A more thorough table explaining everything that is going on as we convert 61 into binary would look like this:

| | | | | | | |
|---|---|---|---|---|---|---|
| Numbered position | 5 | 4 | 3 | 2 | 1 | 0 |
| Power of 2 | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
| Value of power of 2 | 32 | 16 | 8 | 4 | 2 | 1 |
| Did we buy it? | 1 | 1 | 1 | 1 | 0 | 1 |

As with any skill, you should pause and try more examples yourself. The more you practice, the less you have to write down and the more proficient you become. Think of a random decimal number, and convert it to binary. You can use the earlier technique of interpreting binary numbers to check your answers. I recommend that you try both odd numbers and even numbers. Along the way, you should note the following rule of thumb:

- If a number is even, the rightmost bit will be 0
- If a number is odd, the rightmost bit will be 1

This rule can give you a quick check of your answer. For example, if you tried to write the binary representation of 38, but you see that the rightmost bit is 1, you know that something is wrong.

## Octal shorthand

Writing down a binary number can be tedious. So, it is convenient to write a shorthand notation. One such notation is called octal, which means base 8. Here, each digit represents a power of 8, rather than a power of 2. There are 8 possible symbols in octal: 0, 1, 2, 3, 4, 5, 6, 7. The digits 8 and 9 are not possible. Because $8 = 2^3$, each octal digit will correspond to 3 bits in our number.

Converting between binary and octal is pretty simple. The key is to group the binary into clumps of 3 bits at a time. For example:

Octal 671 = 110 111 001 in binary: because 6 = 110, 7 = 111 and 1 = 001

For your convenience, here is a table showing the correspondence between octal digits and their corresponding binary codes. Do not memorize this table. You should be able to perform the conversions yourself.

| Octal digit | Binary representation |
|---|---|
| 0 | 000 |
| 1 | 001 |
| 2 | 010 |
| 3 | 011 |
| 4 | 100 |
| 5 | 101 |
| 6 | 110 |
| 7 | 111 |

Try one yourself: Convert octal 7325 to binary. You should be able to work this out without using the table above as a crutch. First, how many total bits will you need?

Let's go the other way: binary to octal. Group the bits in threes. If there do not seem to be "enough" bits, then pad with extra zeros on the left.

For example, consider the binary 101011100. There are 9 bits here. 9/3 = 3, so we should expect 3 octal digits in our answer. Space out each clump of 3 bits, and convert each clump to its own octal digit.

101011100 binary =
101 011 100 binary =
5 3 4 octal.

Our answer is 534.

Unlike converting between binary and decimal, converting between binary and octal can be done pretty fast no matter how big the number is.

## Hexadecimal shorthand

It turns out that there is a more commonly used shorthand notation called hexadecimal, or just "hex" for short. The word hexadecimal literally means "base 16". It is based on the idea that $16 = 2^4$. Hexadecimal works just like octal, except that each hex digit represents 4 bits rather than 3.

Hexadecimal also means that we have 16 different symbols for each digit. We may use the conventional digit values 0-9, but we need six more. They are:

a = 10, b = 11, c = 12, d = 13, e = 14, f = 15

It may look odd at first, but the letters a-f are in fact used to represent digit values in hexadecimal. It does not matter if you use capital or lowercase letters as long as you are consistent. In practice, lowercase letters are common.

Again, for your convenience, here is a table listing all the hexadecimal digits along with their corresponding binary codes:

| Hex digit | Binary representation |
|---|---|
| 0 | 0000 |
| 1 | 0001 |
| 2 | 0010 |
| 3 | 0011 |
| 4 | 0100 |
| 5 | 0101 |
| 6 | 0110 |
| 7 | 0111 |
| 8 | 1000 |
| 9 | 1001 |
| a | 1010 |
| b | 1011 |
| c | 1100 |
| d | 1101 |
| e | 1110 |
| f | 1111 |

Let's practice converting between binary and hexadecimal.

964 in hex = 1001 0110 0100 in binary, because 9 = 1001, 6 = 0110, and 4 = 0100
d123 in hex = 1101 0001 0010 0011 in binary, because d (13) = 1101, 1 = 0001, 2 = 0010, and 3 = 0011.

Practice yourself:

1. Convert the binary number 111000 to hexadecimal.

2. Convert the binary number 10011111 to hexadecimal.

3. The above hexadecimal table shows the hex and binary representations of the numbers 0-15. What are the hex and binary representations of the numbers 16, 17, and 18?

A note about notations: When we write down a number, and we want to avoid confusion as to what base it is in, it is customary to write the base as a subscript after a number. Therefore, $100_8$ means the *octal* number 100, rather than "one hundred" in decimal or the binary equivalent of 4.

Conversions between decimal and octal/hex

Often, the best way to come up with octal or hexadecimal representations is to go through binary first. It becomes a two-step process: convert from decimal to binary, and then from binary to hex/octal.

For example, how is the decimal number 71 written in octal?

- The Binary Store gives us: 71 = 64 + 4 + 2 + 1.
- Therefore, our binary number is 1000111.
- Grouped as octal, we have 001 000 111 = $107_8$.

Here is an exercise you can try yourself:

1. How are the decimal numbers 8, 9, 10 and 16 represented in octal?

2. How are the decimal numbers 16, 17 and 32 represented in hexadecimal?

3. Here is a little joke: Why couldn't the computer scientist tell the difference between Halloween (Oct 31) and Christmas (Dec 25)?

One final note: The procedures shown above to convert between decimal and binary are quite straightforward to use for small numbers that we have seen. But they become cumbersome for large numbers. For example, to convert 1 million into binary, we would need to know the largest power of 2 that is less than 1 million. And then we'd need to subtract this number from 1 million, and so on. Later, I will show you another way to convert between decimal and binary representations that is less tedious for large numbers and does not rely on determining many various powers of 2.

# PROBLEM SOLVING

A computer system consists of two essential parts: its hardware and software. "Hardware" generally refers to the tangible pieces that you can touch: the actual physical components such as the CPU (central processing unit), memory, I/O devices, and the network. "Software" on the other hand consists of the programs stored on the computer that we want to run. The software is what gives the machine its behavior, and allows it to do useful work (or play). Software includes the operating system, which is responsible for maintaining the hardware in good working order. Besides the operating system is the application software that we use on a regular basis like Web browsers and Microsoft Office.

To illustrate the difference between hardware and software, think of a restaurant. In one sense, when we "go to a restaurant," we are thinking of the building location that houses the restaurant. In another sense, when we are "inside the restaurant," we are enjoying the food and its atmosphere, which are a product of the restaurant as a business. Someday the restaurant (business) might move to a new location (building). It's the same food, employees, clientele and atmosphere, but in a different location. The old location has been taken over by a different business, though its exterior still looks the same.

In this course and in several other courses offered by the computer science department, you learn how to write your own software, so that you can have the computer do exactly what you want and solve your specific problems. We are generally not interested in designing new computer hardware – that is closer to the subject of computer engineering, rather than computer science.

The heart of any computer program is its *algorithm*. The algorithm tells us in English how we go about solving the problem. There are three important features of any successful algorithm:

- Unambiguous: This means that anybody should be able to follow your directions.
- Detailed: The algorithm must be precise in its description of how the input, output, variables, and operators are to be used.
- Deterministic: The order in which the steps are taken must be absolutely clear. When one step is completed, it should be obvious what the next step in the recipe should be.

## Problem-solving strategy

Whenever we wish to solve a problem in computer science, it is important to adhere to a general problem-solving strategy. The following table shows how I like to remember it, with these five steps. Beside each step, I have indicated a common obstacle that could occur at that point in the process.

| # | Problem-solving steps | Common obstacle |
|---|---|---|
| 1 | Understand the requirements of the problem, including its inputs and outputs. | The requirements are vague or ambiguous, or otherwise we don't understand what the problem is asking us to do. In this case, we seek clarification. |
| 2 | Write the solution in English "pseudo-code". This step is the most important, and this is where we need to invest most of our time. | We understand the problem, but we don't know how to solve it! |
| 3 | Write code in a programming language, such as Pascal. | We know how to solve the problem by hand, but we don't know enough of the programming language to adequately convey our solution. |
| 4 | Compile the program. Note that the word "compile" means for the computer to translate your program from whatever language you typed it in, into machine (binary) language, which is the native language of the computer. | After we typed in the program, it does not compile because there are syntax errors. We need to re-edit the program. |
| 5 | Run and test the program. | The program compiles, but when we run it we (sometimes) get a run-time error or the output is incorrect. This probably means we made a mistake back in step 2 because our algorithm is flawed or incomplete. |

Beginners often have trouble with this procedure because they try to do steps 2 and 3 at the same time. It is easier to type the computer code once you already have a good idea on how to solve the problem. Otherwise, you will spend lots of time editing your code and making mistakes.

After a program has been tested and it seems to work for all conceivable inputs, then we can decide to refine it further. For example, we may want to make the output more attractive, or make the input process more user-friendly. Or we may want to generalize the algorithm to solve a larger variety of input situations.

Steps 1-5 above comprise the "problem-solving procedure", in other words the steps you need to follow to solve a problem from scratch. But what if someone has told you how to solve the problem in English, and your job is to transcribe the pseudocode into Pascal? In this case, you are simply following steps 3-5, because 1 and 2 have been done for you already.

A FIRST LOOK at COMPUTER PROGRAMS

Objectives:

- Define a computer program, and describe what it looks like
- Identify certain things that we expect to find in computer programs, such as comments, statements that perform input, calculations, and output
- How to run a program

Why are we interested in creating our own software? The main reason is that we want to harness the computational power of the computer, to have direct control of the machine. Sometimes, existing software that is installed on the computer is not sufficient for us. It doesn't do exactly what we want. Writing our own programs can be extremely useful and even fun for people to use, such as a game, or converting data to an image.

In order for us to create software, we need to learn a computer programming language. There are thousands of programming languages, but only a few have become especially famous in computer science, such as Java, PHP, Python and C++. These languages are machine independent. This means that if you create a program in such a language, it should be able to run on any type of machine. Programming languages have been around for over 60 years, and the modern ones used today have many built-in features that simplify coding. Many common tasks and calculations are pre-defined as part of the language, such as sorting data, opening files, surfing the Web, creating a graphical user interface, etc.

A computer program is a list of instructions written in a computer language that solves some problem. In this course, we will use the Pascal language to write our programs. Pascal is a computer programming language. The text that makes up a computer program is called *source code*, or simply "code" for short. Note that this use of the word code is a mass noun, similar to the word money. In other words, when talking about source code, we always say code in the singular.

When we get into lab, a certain routine is going to emerge. Here are some of the steps.

1. A program is submitted to the computer.
2. We run the program.
3. As the program is running, it may ask for input from you.
4. When the program is finished, we look at its output.

To gain confidence at writing short computer programs, we need to get into the habit of *reading* some computer code. Study carefully the code examples in this short book and ones we do in class. Eventually, you should be able to read several lines of code, and have a precise idea in your mind of exactly what it is doing.

The Pascal language was designed to be a language ideal for learning about computer programming. There are a few reasons why learning a programming language like Pascal will be much easier than learning a natural language like French:

- There are very few words to learn.
- Those words are already in English.
- The grammatical rules are very simple.

So, the challenge of computer problem solving is not learning the language. It is that writing a computer program forces us to think in a way that is logical, organized, and precise – more so than in everyday English.

What does a computer program look like? In essence it works like a recipe: it must list all of the necessary ingredients (data) and instructions (steps) that the computer needs to obey. Cooking may be a good analogy, because it solves an important problem: "I'm hungry!" ☺ What do we generally see in recipes? Well, here is one:

- Brown the beef for 15 minutes. Drain the grease.
- Dice carrot, celery, and onion (otherwise known as mirepoix)
- Cut up and boil 6 potatoes until soft.
- Mash the potatoes.
- Add flour, spices, sauce, and mirepoix to the beef.
- Put the meat mixture into a casserole. Top with potatoes.
- Bake in the oven at 400 degrees for 30 minutes.

Discussion:

1. What dish is this recipe preparing? (In other words, what is the output?) How can you tell?
2. List all the ingredients (input) of the recipe. Why is it important to know all of the ingredients before you start the recipe?
3. Which steps in the recipe imply that we need to continue or wait for something, or do some process repeatedly?
4. At what point(s) in the recipe would it be necessary for a cook to check the progress of a step or to make a decision? What are the decisions or judgments that need to be made?

A simple program will have these elements.

- Comment(s) that explain the program to a human reader
- The program statement
- A list of all named constants and variables we plan to use in the program
- Main program
    - Input statement(s)
    - Calculation statement(s)
    - Output statement(s)

Here is an example program. Programs are usually contained in files, and this one may be called hello.pas. By convention, the name of a file containing Pascal source code will end in ".pas".

```
(* hello.pas
 * This program will ask the user to enter his/her name,
 * and then say hello to that person.
 * This program is very simple.  There are no calculations.
 *)
program hello;

var
  name : string;

begin
  write('Please tell me your name:  ');
  readln(name);

  writeln('Hello, ', name, ', nice to meet you!');
end.
```

Here is a second example program called animals.pas.

```
(* animals.pas
 * Let's illustrate simple I/O and calculations.
 * Add up how many animals on a farm.
 *)
program animals;

var
  horses, pigs, chickens, total : integer;

begin
  write('How many horses?  ');
  readln(horses);

  write('How many pigs?  ');
  readln(pigs);

  write('How many chickens?  ');
  readln(chickens);

  total := horses + pigs + chickens;

  writeln();
```

```
  writeln('You have a total of ', total, ' animals.');
end.
```

The second-to-last statement in the Animals program was a call to the built-in writeln procedure, and we specified no argument. This simply means we want to print a blank line. In fact, in Pascal, it is legal to omit the parentheses from a procedure call if we are not going to pass any arguments. So, the statement `writeln();` could have been written as `writeln;` However, without the parentheses, the word `writeln` looks more like a variable than a procedure. Therefore, I recommend that you include the parentheses anyway even if there is nothing inside. This will make your code more readable.

Computer programs share one thing with poetry: they are arranged in lines. Anything we type between the symbols (* and *) are called comments. Comments are included in a computer program for the benefit of the human reader. The computer actually ignores comments. It is extremely useful for us to put comments in our programs to help us remember the purpose of a program. Also, notice that some of the lines are blank. Pascal allows us to format our code to make the program more readable to us.

Next come declaration statements. The first statement is our program declaration, which gives your program a title. This is followed by the variable declarations. We need to tell Pascal what variables we plan to use in the program. This is analogous to a chef wanting to know what ingredients are necessary in a recipe.

The rest of the program contains the statements that we wish the program to perform. These are enclosed between the keywords begin and end. In the first example above, notice that there are three Pascal statements to execute. Our second example program, animals.pas, contains nine executing statements.

Let's look at another program, one that computes the area of a circle. It can be stored in a file called circle.pas:

```
(* circle.pas
 * This program will find the area of the circle whose
 * radius is supplied by the user.
 * Later on, we could modify the program by having it
 * also compute the circumference.
 *)
program circle;

const
  pi = 3.1415926535;

var
  radius, area : real;
```

```
begin
  write('What is the radius of the circle?  ');
  readln(radius);

  area := pi * radius * radius;

  writeln('The area of the circle is ', area);
end.
```

Again, notice the structure of the program. Let's examine in more detail:

1. The comment. Everything we type between (* and *) is for human eyes only. The computer ignores our comment. The purpose of comments is for us to document the purpose of the program and explain how we wrote it.
2. The `program` statement announces the beginning of the computer program.
3. Constant declarations
4. Variable declarations
5. The main program. Here is where the action is! The "main program" starts with the word "`begin`" and finishes with the word "`end`" with a period. All of the code that we want the computer to execute should appear between this "begin" and "end." Our program contains four executing statements:
   a. A prompt for the user. This is the question that the user will see on the screen.
   b. An input statement for the circle's radius. We use Pascal's built-in `readln` procedure to get what the user types. We specify that the user's value gets put into the variable called radius.
   c. A calculation statement where we specify the formula for the area of a circle.
   d. The output statement. Output is accomplished by using the built-in procedure write or writeln.

Every program has some input and/or output. The programs we are writing are designed to interact with the user. So, input means that the user is typing something that will be used by the program. Output means something is being printed out for the user to see. In Pascal, there are four basic procedures at our disposal to perform I/O:

- read – Obtain a value that was typed at the keyboard. And anticipate that the next piece of input will be entered before the user hits return. This is only useful if we expect the user is entering several values on a line (separated by spaces), which for us will be unusual.
- readln – Read an entire line of input. In other words, assume that the user typed something and then hit return. We will use readln far more often than read.
- write – Print a value or message to the screen. And anticipate that the cursor or next piece of output needs to continue on the same line.

- writeln – Print a value or message, and then go to the next line. We use writeln more often than write.

When using write or writeln, we often have words to print on the screen. All literal text that you want to print on the screen needs to be enclosed in single quotes. For example, we say writeln('hello') to print the word hello. Because the ' symbol indicates the beginning or end of a quoted piece of text, we need a special way to print an apostrophe or single quote. In this case, we type two ' symbols in a row. For example,

```
writeln('Let''s begin.');
```

will print `Let's begin`.

Many of the programs we will write in this course will have this simple structure: input statements come first, then calculations, and then finally output. The Pascal system will execute the statements in the order that they are specified.

One important thing to note is that *the computer has no common sense*. And it doesn't know basic facts about the world unless it's told. The computer may be an impressive machine, but we do have to tell it how to calculate the area of a circle. In a computer program, we must spell out details like this, because the computer is very good at following our literal instructions. If we enter the area formula incorrectly, the computer will never warn us we made that kind of mistake.

Well, now you have seen some computer programs. Let's examine them closely. What do they contain?

- Statements – A statement performs one step of the program. Each "line of code" usually has one statement. There are several types of statements in Pascal that we will study. Sometimes, we may use the words "instruction" or "command" interchangeably with the word "statement". However, in later courses, you will see that there is a nuance in meaning between these words.
- Variables – A variable is a place in memory to hold information, such as a person's name or the area of a circle. Every variable has a name, and the name of a variable is called an identifier. Variables have a type, such as integer, real, boolean (true/false) and string (i.e. text).
- Constants – Some constants have a name such as "pi" or we type a literal number such as 5.
- Expressions – often the stuff you see in parentheses or on the right side of an := operator.

The way that we type out programs should make them easy to read.

- Whitespace – This includes spaces, tabs and newlines. They are generally ignored by the Pascal system. But they are necessary for human readability. For example, please put a space on either side of an operator like + or :=.
- Indentation – Sometimes, statements need to be grouped together, and we convey this by indenting. Pascal does not require it, but it is highly recommended. For the code examples you see in this course, pay attention to how the code is indented, and try to mimic that in your work.

- Lines of code should not exceed 80 characters. If you have a very long statement, it is fine to continue it onto the next line. If you do so, make sure that the continuation is indented far enough so that it doesn't appear to be its own statement.

At this point, you may notice that some words we use are also found in ordinary mathematics. For example, in algebra, one talks about constants, variables, and expressions. One major difference between computing and algebra terminology has to do with the = sign. In algebra, the = sign signifies an equation. An equation is a fact stating that two numbers are the same. But in computer programming, we don't use the word equation, and we don't work out equations as you would in algebra. A computer program is a list of instructions for the computer to perform, rather than simply a list of facts about numbers.

The purpose of this class is not to teach you everything about Pascal – rather, just enough Pascal so that you can solve some useful programs using the language.

*While you are leaning Pascal, it is always an excellent idea for you to experiment with the code on the computer. Type in some short programs like the ones you see here. Make modifications to them. Enhance them. Observe how the output changes. Observe what kind of errors you encounter. Be curious and use your imagination.*

Discussion:

1. What is a computer program?

2. What does a computer program look like?

3. Does a program resemble anything you have seen before?

4. What little things do we see in a computer program?

5. How can we tell if it works correctly or not?

6. If incorrect, what could have gone wrong?

7. What different kinds of statements have we seen so far?

Variables

The purpose of a variable is to keep track of a value that we need in our program. Every variable needs a name. The technical term for a variable name is an *identifier*. An identifier is usually just a single word that refers to a variable we want to maintain in our program. Pascal has specific rules on identifiers.

- It must begin with a letter or underscore.
- After the first character, the identifier may include letters, digits, or underscores.
- Identifiers are not case sensitive.
- It may not be a keyword that has a predefined meaning in Pascal. The following table shows a list of words that you may not use for your variables. You do not need to memorize this list! If you attempt to declare a variable with any of these names, you will get a syntax error message.

```
and, array, asm, begin, case, const, constructor, destructor,
div, do, downto, else, end, file, for, function, goto, if,
implementation, in, inherited, interface, label, mod, nil, not,
object, of, operator, or, packed, procedure, program, record,
repeat, set, shl, shr, string, then, to, type, unit, until, uses,
var, while, with, xor
```

- It also may not be a "standard identifier". Some words are occasionally used in Pascal because they perform some useful purpose. Example of such words are "true", "false", "integer", and "read". If you declare a variable with one of these names, you won't get an error message, but your program may exhibit odd behavior. Computers don't like ambiguity. Every word in your program needs to have a single meaning. For example, it would be a very bad idea to declare a variable with the name `read`. If you did so, Pascal would forget how to use the built-in read procedure! So, if you are using a built-in identifier, do not also use it as one of your variable names.

Variable declarations are preceded by the keyword `var`. Only type "var" once, regardless of how many variables you want to declare. Then, the format of a variable declaration is as follows.

*Variable name : type;*

If there are several variables of the same type, it is convenient to declare them all on the same line. For example, to declare three variables of type real we could say

```
length, width, height : real;
```

You should also be careful to avoid declaring the same variable twice. If you accidentally do so, the compiler will give you an error message.

Besides these basic rules, in the computer science community, we have a general rule of style that also advises us that our variable names should be meaningful. To make your program easier for people to understand, the name that you give to a variable should be a strong clue as to the type of value that you wish to store. If you want a variable to count something, then call it something like `count`. Some quantities are naturally meant to be integer, such as a number of people. Money should be considered a real number. And text is a string.

Nevertheless, Pascal allows you to write confusing code. For example, you could declare a variable called x. What does x mean? The variable name x is so generic that it could stand for anything. Therefore, it is usually considered poor programming style to have relatively meaningless variable names, especially one-letter names. A good variable name would be a word or short phrase. Nevertheless, if you intend the keep track of the x-coordinate of a point, or you are doing some purely algebraic or calculus problem, a variable name like "x" could be appropriate. *Coming up with meaningful variable names is an important step in computer problem solving*.

Occasionally, we find it useful for a variable name to consist of more than one word. For example, suppose your program needs to keep track of the radius and mass of both a planet and its moon. In this case, a variable called `moon` or `mass` would be ambiguous, and we have to resort to using 2-word identifiers. Good variable names in this case would be:

```
PlanetMass
PlanetRadius
MoonMass
MoonRadius
```

Note the convention of capitalizing each word of the variable name. Remember that Pascal is not case sensitive. So, `planetmass` and `PlanetMass` refer to the same variable. We capitalize the individual words in the variable name to make our code more readable. Alternatively, we could use lowercase letters throughout, and separate the words of an identifier with an underscore, like this: `planet_mass`. But note that the underscore counts as one character in the identifier, so that `PlanetMass` and `planet_mass` do not refer to the same variable.

## Constants

Sometimes, we have a value in our program that we know will never need to change during program execution. For example, Social Security numbers always have 9 digits. The value of π is constant. The sales tax may be 7%, and we don't want the user to change this value.

So, in addition to variables, Pascal allows us to declare constants at the top of our program, before the variables. If we have any constants to declare in our program, we preface them with the keyword const, and then specify each constant with a name and a value as follows:

```
const
  rate = 1.07;
  TouchdownValue = 6;
```

<u>The assignment statement</u>

This is the most important statement type. It allows us to initialize or change the value of a variable. Assignment statements appear very frequently in computer programs. This type of statement uses the := sign, which is formally called the assignment operator. We read it as "colon equals." The format of an assignment statement is as follows:

< Variable > := < Expression > ;

To the left of the := sits the variable that we wish to define or update. To the right of the := is the value or the expression that we want to store in the variable.

An aside about my notation here: the symbols < and > when used around a word like < Variable > are called *angle brackets*. They are used to enclose the name of some category. For example, in a grammar book, you might see angle brackets used when defining the format of a sentence, like this:

< Subject > < Verb > < Direct Object >

Now, here are some examples of assignment statements:

```
rate := 1.07;
name := 'Mickey Mouse';
NumPeople := 24;
root := -b + sqrt(b*b – 4*a*c) / (2 * a);
area := height * (top + bottom) / 2.0;
```

When we use the assignment operator :=, we are issuing a command to the computer. For example, "`rate := 1.07;`" says that we want the value 1.07 to be stored in a variable called rate. This kind of statement can be translated in English as "let `rate` be 1.07" or "assign `rate` the value 1.07".

There are two possible effects of an assignment statement:

1. If the variable has not yet been given a value, give it an initial value.
2. If the variable already has a value, replace its current value with a new value.

For example, when we say "x := 5", this has two possible effects. If the variable x has never yet been used in the program up to this point, then the computer will put the value 5 in it. But if the variable x already had a value stored in it, its old value is overwritten with the value 5. The old value is then discarded. The computer will not warn you if a variable in your assignment statement already exists or not. The computer will assume that you know what you are doing.

You need to assign a value to a variable before you can use it. This is because when you first declare a variable, its initial value is technically undefined. In practice, there may be some initial value in the variable, but may be just a garbage value left over from a previous program.

While running your program, the computer never forgets what your variables contain. When we assign a value to a variable, as in x := 5 for example, this value is stored in the variable for the rest of the program, unless we overwrite this value later. In other words, the computer will not "forget" or lose the values that you assign, until the program is finished. Upon program termination, all of its variables disappear. The memory allocated to your program's variables can be given to other programs running on your system.

For example, suppose a computer program has these statements:

```
a := 7;
b := a + 1;
a := 4;
```

And then the program does a thousand other things. Assuming that the values of a and b are not reassigned again, at the end of the program, the values of a and b will still be 8 and 4, respectively. Just because you haven't used a variable for a long time in a program does not mean the information becomes stale or forgotten.

Discussion:

1. Assume that the variables a, b, c, and d have been declared to be of the integer type. What values are contained in the variables a through d after the following code executes?
   ```
   a := 3;
   b := 4;
   c := 5;
   d := c;
   c := a;
   b := b + a;
   a := d;
   ```

2. Suppose that x, y and z are real variables. What values are contained in these variables after the following code executes?
   ```
   x := 9.0;
   y := 2.0;

   z := x;
   x := y;
   y := z;
   ```

3. How should we respond to the following comment made by a former student?
   "Computers make no sense. For instance, they allow you to say n := n + 2. Obviously, that is a false statement. How could n possibly be equal to 2 more than itself?"

Variable types

At this point, there are three variable types that you need to be aware of. They are called `integer`, `real` and `string`. They are for integers, real numbers, and strings, respectively. A variable may be of any one of these types. When you declare a variable in Pascal, you must specify its type. Although you are free to change the value of a variable throughout the life of the program, you cannot change the variable's type. Once an integer, always an integer. Recall that the general format of an assignment is as follows:

< variable > := < expression > ;

In such a statement, the < variable > and the < expression > generally need to be of the same type.

Literal constants (e.g. 'hello', 2.5, and 16) automatically have a type. A string is easy to spot by its quotation marks. The difference between a `real` and an `integer` is that a `real` constant will always have a decimal point with digits on both sides. Thus, 4 is an `integer`, while 4.0 is a `real`. Both these numbers are equal, but they are of different types. In fact, you will even sometimes see number inside a string: '4' is a `string` rather than an `integer` or `real` number.

There are some simple rules concerning the type of an expression.

1. When combining expressions of the same type, the resulting type is unchanged. For example, adding integers will result in an integer.
2. When combining an integer and a real, the result will be a real. For example, 2.0 + 3 will result in 5.0, not 5.
3. The / operator performs real number division, and its result will always be a real. If you want to perform divisions on integers, (and truncate any remainder that might result), then you use the word div. So, remember: / means real division, and div means integer division. For example, 29.0 / 10.0 equals 2.9, while 29 div 10 equals the integer 2.

Pascal provides a couple of useful functions that allow us to convert a real number to an integer. They are `trunc` and `round`. The function `trunc` means to truncate, meaning truncate the fractional part of the number, or "round towards zero". On the other hand, `round` will round to the nearest integer.

Here are some examples of using `trunc` and `round`. What do you notice about their behavior?

| | | |
|---|---|---|
| trunc(5.1) = 5 | round(5.1) = 5 | round(7.5) = 8 |
| trunc(5.8) = 5 | round(5.8) = 6 | round(8.5) = 8 |
| trunc(–5.1) = –5 | round(–5.1) = –5 | round(–7.5) = –8 |
| trunc(–5.8) = –5 | round(–5.8) = –6 | round(–8.5) = –8 |

In Pascal, mathematical expressions are evaluated just as in ordinary arithmetic. We follow the same order of operations:

1. Parentheses first
2. Then, multiplication and division (operators * and div and /)
3. Finally, addition and subtraction (operators + and –)

Pascal has one more arithmetical operator that will be very useful for us. It is the word `mod`, which we place between two integers, as in the expression `14 mod 3`. The purpose of the mod operator is to calculate the remainder of integer division. For example, when we do long division on integers, we compute a quotient and remainder. When we divide 14 by 3, the quotient is 4 and the remainder is 2. In Pascal, we would say that 14 div 3 is 4, and 14 mod 3 is 2. The mod operator has the same level of precedence as the multiplication and divide operators.

Besides the precedence of operators, we need to know that the evaluation proceeds from left to right. For example, 8 – 3 + 2 is interpreted to mean (8 – 3) + 2. This property is known as "left associativity".

Discussion:

1. How are variables in Pascal different from the variables that you saw in high school algebra?

2. What data types are available?

3. What do these operators do? * / div mod

4. What is the difference between `trunc` and `round`?

5. Show how the mod operator can be used to work out the following problems:
   a. What day of the week is it 30 days from today?
   b. Is this year a leap year?
   c. What is the minimum number of pennies we need to pay 58 cents?

6. By hand, work out this expression: 30 – 3 * 3 + 8 div 3 * 3 * 10

7. Write a program that finds the area of a rectangle, using input given by the user.

<u>Large integers</u>

In Pascal, the integer type is usually limited to a 16-bit representation. As a consequence, the range of possible values for an integer variable or expression is –32,768 to +32,767. For many purposes, this range is sufficient. However, if you suspect that there is a possibility that the integer values in your program may exceed 32,767, then you need to use a larger integer type. A common choice is `int32`.

As its name suggests, this is a 32-bit integer, which permits values between approximately positive and negative 2 billion.

## Formatted output

We use the built-in procedures write and writeln to accomplish output. Usually, we want to write an entire line of output at a time, so we tend to use writeln more frequently. The only difference is that, as the name suggests, writeln will end its output with an invisible newline character so that future output will begin on the next line. Just as when we hit return when we type.

Write and writeln will print whatever we supply inside parentheses. We can print numbers, text, several of each, or any combination. We can print constants, variables, and expressions. If you want to print multiple items in a single call to write or writeln, these must be separated by commas. For example, suppose that `length` and `width` are integer variables. We can say:

```
writeln('The dimensions are ', length, ' by ', width, ' feet.');
```

The issue of formatted output becomes important when printing real numbers. Unfortunately, by default, Pascal prints real numbers in scientific notation. To print a real number the ordinary way, we must specify two quantities:

- How many total characters we want to print, including the digits before the decimal point, the decimal point itself, and the digits after the decimal point. This is called the “field width”.
- How many digits of precision, namely the digits after the decimal point.

For example, let’s say we have the variable `price` equal to 123.45. To print this number, we observe that we need 6 total characters, and 2 of these to the right of the decimal. So, we could say:

```
writeln('The price is $ ', price : 6 : 2);
```

The two formatting parameters are separated by colons. Often we are not sure in advance how much space to allocate to a number. In this case, we ask ourselves what the maximum could be. If we are printing amount of money, we are confident that we want 2 digits of precision. Then, if we assume that everything is under a million dollars, the total field width would be 6 + 1 + 2 = 9 characters. So we would print with the code `price : 9 : 2`.

We can also format the printing of integers. But for integers, there is no precision, only a field width. The effect of supplying a field width is to right-justify a number. For example, if a = 9, b = 45, and c = 102, then:

```
writeln('a = ', a : 3);
writeln('b = ', b : 3);
writeln('c = ', c : 3);
```

will display

```
a =   9
b =  45
c = 102
```

Errors

In computer programming, we classify errors into three categories. The word "bug" is also used as slang to refer to any of these mistakes.

Syntax error – This means that you have entered a statement that the computer does not understand. If the machine is not 100% certain of what you are trying to do, it will give you this kind of error message, and refuse to run the program. Even the most trivial typos can give rise to a syntax error.

A syntax error is usually the easiest kind of mistake to fix. The word syntax basically means grammar, so a syntax error typically means that your program contains at least one statement that violates the rules of the Pascal language. For example, misspelling the name of a keyword, missing or extra operator or punctuation symbol. If your program contains a syntax error, the Pascal system will immediately tell you about it when you try to run your program. Because the computer does not understand your program, it won't even start to run it.

Run-time error – This means that the computer understands your program, but it aborts while it's running because you are attempting to perform some operation that is impossible. In this case, the program will immediately halt during execution, and you will see an error message.

Common sources of run-time errors include trying to open a file that does not exist, dividing by zero, taking the square root of a negative number, etc. As the name suggests, a run-time error occurs while the program is running. As with syntax errors, the Pascal system will tell you what kind of error you have, and where in the program it thinks the error has occurred.

Logical error – This means that the program runs to completion, but the output is incorrect.

Logical errors usually result when we mistype a formula or when we print the wrong value. Logical errors are usually the most difficult kind of error to fix because the computer does not give you an error message. The machine is simply unaware that anything might be wrong. You must logically figure out where the error came about.

Remember: An error is simply a mistake in a program. Computer programs must be written with great care because they are brittle. A trivial change in the code can make a big difference in how the program behaves. It is very easy to make a mistake. One small change can fix or ruin a program. Don't make it a rush job. For example, when editing, it is easy to omit a statement, duplicate a statement, type statements out of order, or indent statements improperly.

## Algorithm

Before we move on, I think it would be a good idea to explain one very important concept that we will need throughout the course. It is the notion of an algorithm.

An algorithm is a list of instructions written in English that explains how to solve a problem.

Does this sound familiar? How does this definition compare with the definition of a computer program that we saw earlier? They are both lists of instructions; they are both recipes for a solution. But a computer program must be written in a computer programming language, while an algorithm is written in our human language.

When we solve problems, we are basically in the business of writing algorithms. Besides being correct, an algorithm should also be:

- Unambiguous
- Detailed
- Deterministic: As we “control” the CPU, we should be clear where the next step is, and when we are finished.

Later on, we will practice creating algorithms. It’s a very important skill, even more important that the nuts and bolts of any programming language.

At this point, you already know how to write some simple computer programs. From now on, we will learn a little more about Pascal in order to add to our arsenal. For example, you will be enlarging your knowledge about the different kind of statements that can appear in a Pascal program, and the different types of variables that you can use.

## Some advice

The best way to learn how to solve problems is to practice. But it doesn’t hurt to take a break for a few minutes and listen to some advice. ☺

To motivate a solution to a new problem, it sometimes helps to look at existing example solutions. After a while, as you accumulate some experience, you will begin to say to yourself, “This problem looks familiar.” Most problems are not solved entirely from scratch. You can adapt a technique that you saw from an earlier problem. It’s just like making sandwiches. If you know how to make a peanut butter and jelly sandwich, it’s not hard to figure the recipe for a peanut butter and banana sandwich. And the procedure for a turkey sandwich is almost the same as for a ham sandwich.

Use built-in features of Pascal to simplify your solution. You can search the online documentation, and maybe you will find a very useful function. We have seen examples of this already. For example, the built-in string functions.

One major skill in computer problem solving is being able to read a problem description, and identifying the major "nouns" and "verbs." The nouns often become the variables in the program. The verbs could be various operations or functions.

It has been said that any skill takes about 10 years to master. In a college course you only have a few months. You won't be expected to solve every possible problem at this stage in your career. So, be patient with yourself and have some fun. Learning to write computer programs is a skill like many others that you acquire. And it's a skill that finds wide application in almost every field.

Problem solving is not easy. There is no formula that works in all cases. Difficulties can arise at any stage in the process. Ask for help if you get stuck.

It's easier to find a mistake in the (English) design of a program than in the (Pascal) implementation. This is why we spend so much time in step #2 in the problem-solving procedure. It's a common mistake for people to rush to step #3 and type the code before having a complete algorithm.

Input/output (I/O) is an important part of computer programs. To create an effective program, we sometimes have to use the right kind of I/O.

- Examples of input include: numbers, text, files, a URL, a button that the user can click, and image and sound
- Examples of output include: numbers, text, files, images and sound

The output is what the user sees when running your program. You need to take care that the output is in the exact specification that the user expects. For example, suppose you want to print an amount of money. An output of $3.57777777 does not look good. People appreciate output that is formatted attractively.

When you write your own algorithms, it is not necessary to number your steps. After all, as you compose your algorithm, you might suddenly realize that you need to insert a step between two existing steps, and it would be inconvenient to have to renumber the steps.

Discussion:

1. What is the difference between a computer program and an algorithm?

2. For each of the following quantities, indicate the appropriate data type. Your choices are integer, real, boolean and string.
   a. the length of a fish in inches

b. the number of people on a train
c. whether a job applicant has a college degree or not
d. the exact number of gallons of gasoline consumed after driving 100 miles
e. the user's complete street address
f. the number of pounds of rice used in a recipe
g. the number of television sets in a house

3. The user is going to the post office, and would like to buy postage for a certain number of first-class letters and post cards. Ask the user for the number of each desired, and calculate the total cost of postage. Assume that first-class stamps cost 58c and post cards are 40c each.

4. Show how we can print a real-number variable as a dollar figure as high as $ 9999.99.

5. Write a computer program that asks the user for the dimensions of a rectangular swimming pool in feet. The program will then determine the total volume of water in gallons. Assume that there are 7.8 gallons of water per cubic foot. Print the result to one decimal place.

6. Suppose you are in charge of the pepperoni slices at a pizza factory. Each pepperoni pizza requires 30 slices of pepperoni. Your boss says that if the total number of slices you make is not exactly a multiple of 30, then you may eat the leftover slices that are not enough to put onto the last pizza. (The last pizza will be sold as just a cheese pizza.) Today you have cut N slices of pepperoni. How many pepperoni pizzas can you make, and how many slices will you eat?

7. Solve a quadratic equation. Ask the user for the values of a, b, c in $ax^2 + bx + c = 0$, and determine the two answers. You may assume that the roots are real. Use the built-in function sqrt to take square roots.

8. Suppose a company compiles a report on all of its employees. The report is formatted so that there are always 4 employees listed per page. The pages are numbered sequentially, starting at 1. The employees are number sequentially, starting at 1, and appear in ascending numerical order throughout the report. Write formulas that will determine the following:
   a. If there are n employees, how many pages will be in the report?
   b. On which page will we find the report for employee number n?
   c. Page n contains the reports for which employees? In other words, give the lowest and highest numbered employee on page n.
   d. Let's generalize these formulas. How would your formulas change if we formatted the report to have P employees per page?

9. Ask the user for a car's city and highway mileage (miles per gallon), and use this information to compute the average MPG. The definition of "average" MPG assumes that 55 percent of all miles are city miles, and the remaining 45 percent are highway miles. Hint: Assume that we drive 100 miles. Therefore, we drive 55 miles in the city and 45 on the highway. The average MPG is the total

number of miles driven (100) divided by the total number of gallons consumed in the city plus the highway. In your solution, you need to calculate the number of gallons consumed driving 55 miles in the city, and the number of gallons consumed driving 45 miles on the highway.

10. Suppose you arrive home after a long day at work, and you notice that a power failure had occurred while you were away. You would like to know when the power failure took place. Based on the clocks in your house, it may be possible to figure it out! Assume that inside your house, you have two electric clocks. One is analog. When the power goes out the clock simply stops, and when power resumes the clock continues counting time where it left off. The other clock is digital. When the power goes out it erases its time. When power comes back, it starts counting time over at 12:00.

    Assuming that the power failure had started less than 12 hours ago, devise an algorithm that will determine when the power failure started and ended, given three pieces of information: the time that you arrive home, the time displayed on the analog clock, and the time displayed on the digital clock. You know the current time because your watch was unaffected by the power failure. As an example, what could you conclude if you arrive home at 7:00, the analog clock says 5:50 and the digital clock says 4:30?

IF STATEMENTS

As we have already seen, a computer program is a list of instructions. In what order are the instructions performed? By default, in the order that we specify. In other words, if a computer program has 3 statements, then the computer will perform the first statement, followed by the second statement, and finally the third statement.

To be able to solve more useful problems, we need to write computer programs that can do more than just sequential execution. In particular, two capabilities are needed:

- To make choices; to take a fork in the road. In other words, to follow one path instead of another. This includes the ability to skip some statements that we sometimes don't need to perform.
- To repeat some steps. In other words, sometimes we want a "loop."

In computer science, we usually use the term "control structures" when referring to these two capabilities. We will practice making choices and repetition quite a lot, because they are so important in problem solving. Along the way, we will learn some Pascal keywords to handle these situations. The words `if`, `else if` and `else` are used for making choices, and the words `while` and `for` are used for repetition.

Before we dive into our discussion of control structures, I want to review some basic facts about computer programs:

- A program needs to have output. (Otherwise, how would we know that it works correctly?)
- Output is usually the result of some calculations we need to do in the program.
- A program's calculations are usually based on input.

This is why we tend to say that a computer program has 3 important parts: input, calculations and output. We can solve a greater variety of problems if we allow our programs also to *make choices*, to ask questions about either the input values or the results of calculations. A program can decide which path to follow based on whether the answer is yes or no.

## The if-statement

Sometimes, we need our program to make a decision. In other words, we want our program to do something based on what values are in our variables. Here is the basic format of an if-statement:

```
if <condition> then
  (* steps to perform if the <condition> is true *)
```

Optionally, we can include an "else" portion, so that the entire if-statement looks like this:

```
if <condition> then
  (* steps to perform if the <condition> is true *)
else
  (* steps to perform if the <condition> is false *)
```

Every if-statement has a condition to test. An if-statement is essentially asking a yes/no question, usually about the value in a variable. Comparisons are done with a *relational operator*. There are 6 relational operators that we can use in Pascal:

| Relational operator | Meaning |
|---|---|
| < | is less than |
| > | is greater than |
| <= | is less than or equal to (i.e. is at most) |
| >= | is greater than or equal to (i.e. is at least) |
| = | is equal to |
| <> | is not equal to |

Please note the difference between = and :=. The := operator is used in an assignment statement. The = is a relational operator. To illustrate, consider the following examples:

- x := 5 means that we command the computer to put the number 5 into the variable x.
- x = 5 is a question asking the computer if x is equal to 5 or not.

A comparison is an example of a *boolean* expression. Boolean means the value is either `true` or `false`. It turns out that boolean is actually a data type in Pascal, along with integer, real, and string. So now you know four data types. With the boolean type, it is possible to assign a variable to be `true` or `false`. Later, we will see examples where this is especially useful.

Since an if-statement expects a boolean expression (to answer a yes/no question), and Pascal allows a variable to be boolean, this means that the boolean expression used in an if-statement can be just a boolean variable alone. For example, let's suppose that ValidInput is a boolean variable. Then, it is legal in Pascal to write an if-statement starting with:

```
if ValidInput then
```

although it would be perfectly fine to say:

```
if ValidInput = true then
```

Similarly, to inquire whether a boolean value is false, it suffices to use the word "not" as follows:

```
if not ValidInput then
```

in place of

```
if ValidInput = false then
```

<u>The body</u>

After we specify the if-statement's condition, we write the "body" of the if-statement. This consists of the steps we wish to perform if the condition is true. If there is an "else", we specify a second body to contain the steps to perform if the condition is false.

A "body" may be a single statement. Or, it may be any number of statements between the words "begin and "end". In Pascal, if you write "begin" and "end", and place several statements in between, this is called a "block of statements", and a block of statements can go in the place of a single statement. For example, we can write:

```
if n = 0 then
  writeln('The number is zero.')
else
  begin
    writeln('The number could be positive.');
    writeln('Or it could be negative.');
  end;
```

Notice that the word end is followed by a semicolon, not a period. The only time you see end followed by a period is at the end of the entire program.

This if-then-else statement has two bodies. The first body appears immediately after the word `then`. The second body occurs after the word `else`. In this example, the first body contains 1 statement. But the second body contains 2 statements.

If a body needs to contain more than 1 statement, then the begin/end enclosure is required. However, if the body contains just 1 statement, then the begin/end is optional. Therefore, it is always safe to use `begin` and `end`.

Occasionally, you might be working on an unfinished program, and you aren't sure what statements to put in the body. In this case, it is safe to have an empty body. A minimal body would consist of an empty statement, which you could represent as just a single semicolon. Or you could type `begin` and `end` but have no statements in between. But to leave a body completely blank (e.g. typing nothing between the words `then` and `else`) may lead to confusion.

Using the words "and" and "or"

When using relational operators to ask a question about a variable, it's even possible to ask multiple questions, combined with the words `and` or `or`. For example, an `if` statement could perform two comparisons like this: `if (answer = 1) or (answer = 3) then …`

This is sometimes called a "compound condition" because it consists of two or more conditions separated by the words `and` or `or`. Here, this first condition is answer = 1, and the second condition is answer = 3. The purpose of the compound condition is to test whether the value of answer is either 1 or 3. Please note that when typing out a compound condition like this, all of the individual conditions that make up the compound condition need to contain a relational operator. For example, one mistake that beginners sometimes make is to write an `if` condition like this:

```
if month = 'September' or 'April' or 'June' or 'November' then
```

The intent here is to create a compound condition with four parts. However, the string 'April' by itself is not a condition. To write this correctly, we have to spell out exactly what we want as 4 complete conditions, as follows.

```
if (month = 'September') or (month = 'April') or (month = 'June') or
   (month = 'November') then
```

We often need to use the word `and` if we want to ask if a variable is between two values. For example, we might want to know if x is an integer between 50 and 100. We could write the condition as follows:

```
if (x > 50) and (x < 100) then
```

As you know, an inequality expression can be turned around as long as we remember to reverse the inequality sign. So, x > 10 has the same meaning as 10 < x. However, when comparing a variable to a constant, it is customary to put the variable on the left and constant on the right. Nevertheless, it is legal to write the above example as:

```
if (50 < x) and (x < 100) then
```

Note that if we wanted x to be between 50 and 100 *inclusive*, we would write

```
if (50 <= x) and (x <= 100) then
```

Here is an example of a program or program fragment that uses if-statements.

```
(* input *)
write('Enter an integer:  ');
readln(n);

(* Tell user if it's positive. *)
```

```
if n > 0 then
    writeln('Your number is positive.');

(* Tell user if the number is odd or even *)
if n mod 2 = 0 then
  writeln('Your number is even.')
else
  writeln('Your number is odd.');
```

It is important to take note of some syntax requirements with if-statements.

1. Don't forget the word "then" after you write the condition.
2. If your if-statement contains an else clause, do not put a semicolon before the else.
3. If the then-clause or else-clause contains more than one statement, these must be enclosed in a begin & end block.

In addition to these syntax requirements, it is strongly recommended that you indent your code by 2 spaces as in these examples. You are already familiar with the concept of indentation when grouping information in a hierarchical manner. Think of an outline, or the bullets of a presentation. For example, suppose you are studying the notes of a meteorology lesson. A simple outline might look like this:

1. Temperature
2. Precipitation
    a. Rain
    b. Snow
    c. Hail
3. Wind

Notice that section 2 has subsections a, b and c. When we finish subsection c and want to proceed to section 3, we unindent back to the level of the Arabic numerals. Pascal code is indented and unindented in a similar way, except that we do not label our statements with numbers or letters.

Let's practice writing if conditions for the following situations. In the following exercises, assume that we have a positive number held in an integer variable called n. In each case, you should write a line of code that introduces an if-statement. It will have this format, and all you have to do is to supply what goes in the blank.

if ____________________ then

1. The number is between 5 and 25 inclusive.
2. The number is less than 12 or equal to 95.
3. The number is even and less than 100.

4. The number is between 10 and 20, or between 80 and 90.
5. The number ends in 7.
6. The number ends in 67.
7. The number is neither 13 nor 17.
8. Now, let's suppose we have two numbers x and y. Write the condition that says that y is positive and x is strictly between 0 and y.

A very useful application of the if-statement is determining if a year is a leap year. The traditional Julian definition of leap years is pretty simple. We just need to verify that the year number is divisible by 4. Here is what the code would like:

```
if year mod 4 = 0 then
  writeln('Leap year')
else
  writeln('Not a leap year');
```

Unfortunately, we no longer use the Julian definition of leap years. It turns out that to be more astronomically accurate, we need to use what is called the Gregorian definition of leap years. The rules to qualify as a leap year are as follows:

- If the year ends in 00, then the year must be divisible by 400.
- If the year doesn't end in 00, then it must be divisible by 4.

Exercise:

1. Using the Gregorian definition, write Pascal code that will print "Leap year" or "Not a leap year", as appropriate, based on the value of the `year` variable.

2. Give an example of a year that qualifies as a leap year under the Julian definition but not under the Gregorian definition.

3. Let's calculate someone's weekly wage. Ask the user for the hourly rate and number of hours worked. Be sure to pay overtime if the number of hours worked is over 40: for each additional hour the hourly rate needs to be 50% higher.

4. Ask the user to enter two numbers. Determine which number is larger, and by how much. For example, if the user enters 2 and 6, tell the user that the second number is larger, and that the difference is 4.

5. Suppose we have an integer variable NumDogs. We want to print a sentence such as `There are 5 dogs`. But if the value of NumDogs is 1, we need to print the word `dog` in the singular. Show how we can handle this.

6. Explain what is wrong in each of the following code fragments.

```
write('Enter the number of people:  ');
readln(NumPeople);
if NumPeople := 0 then
  writeln('You entered zero people.');

-----------------------------------------
if letter = a or e or i or o or u then
  writeln('The letter is a vowel.');

-----------------------------------------
write('Do you want to quit the game?  (y/n):  ');
readln(choice);
if (choice <> 'Y') or (choice <> 'y') then
  writeln('Great!  The game will continue...');
```

If-statements using multiple choices

Just as it's possible for a program to need several variables, it's also possible that we need more than 2 alternatives with our if-statements. How would we handle 3 or more? For example, given an input value, we may need to test if it is positive, negative or zero. Another example is converting a numerical grade to one of 5 different letter grades according to a grading scale.

You already know that we use the words `if` and `else` with if-statements. The way to allow for additional choices is by using `else if`. Here is the general format of an if-statement that uses else if:

if <condition 1> then
   (* steps to perform if condition 1 is true *)
**else if <condition 2> then**
   **(* steps to perform if condition 2 is true *)**
(* You can have as many `else if`s as needed *)
else
   (* steps to perform if all conditions were false *)

For example, here is how to test for positive / negative / zero:

```
if value > 0 then
  (* do positive case *)
```

```
else if value < 0 then
  (* do negative case *)

else
  (* At this point, we know value has to be zero! *)
```

And here is how a grading scale can be used to assign letter grades. If the criteria are 90, 80, 70 and 60, the code would look like this:

```
if grade >= 90 then
  letter = 'A'
else if grade >= 80 then
  letter = 'B'
else if grade >= 70 then
  letter = 'C'
else if grade >= 60 then
  letter = 'D'
else
  letter = 'F';
```

Notice again that we never put a semicolon before the word else.

Please note that when you write an if-statement with several alternatives, only <u>one</u> of the bodies can execute. Suppose that grade equals 75. Even though 75 > 70 and 75 > 60, the value of letter is 'C' not 'D', because we execute the body corresponding to the <u>first</u> condition that we encounter that is true. Once we execute the appropriate body, we exit the entire if-statement.

The above grading scale can be adapted to other purposes as well. For example, hurricanes and tornadoes are often classified by a five-category scale according to their maximum wind speed (Saffir-Simpson for hurricanes and Fujita for tornadoes).

Discussion: Modify the grading scale code above to account for plus and minus grades. For example, let's assume that grades ending in 0 or 1 get a minus, and grades ending in 8 or 9 get a plus. What other assumptions should you make?

Let's look at an example of an algorithm that uses "if". Convert time zones. Given the time here, add 9 hours to output the corresponding time in Moscow.

Algorithm:

1. Ask the user to enter the hour, the minute, and whether it is a.m. or p.m.
   Throughout this algorithm, the minutes will be unchanged.

2. Convert this civilian hour to a military hour:
   If it's 12 a.m., the military hour is 0
   If it's 12 p.m., the military hour is 12
   Otherwise, if it's p.m., the military hour is 12 + the civilian hour.
   And if it's a.m., the military hour is the same as the civilian hour.
3. Ready for the conversion. Add 9 to the military hour.
   After doing so, if the military hour is 24 or higher, subtract 24 because it is referring to a time in the next day.
4. Convert the hour back to civilian format. In each case, print the minute between the hour number and the a.m./p.m.
   If the military hour is 0, output 12 a.m.
   If the military hour is 12, output 12 p.m.
   Otherwise, if the military hour > 12, subtract 12 and say "p.m."
   Else, print the military hour unchanged, and say "a.m."

Exercise:

1. How would this algorithm change if we wanted to convert to a time zone in the other direction? For example, Los Angeles is 3 hours behind.

2. How would this algorithm change if we wanted to convert to a time zone that is not a whole number of hours different from ours? For example, the time in Newfoundland is 1.5 hours ahead.

Here are two complete programs that feature if statements.

```
(* istriangle.pas - Given the lengths of 3 line segments, see if they can
 * form a triangle.  The rule is that the longest has to be shorter than the
 * sum of the other two sides.  But first, we must determine which line
 * segment is the longest.  This program illustrates the if-then statement.
 *)

program istriangle;

var
  a, b, c, longest, leg1, leg2 : integer;

begin
   (* Input *)
  writeln('Let''s see if you really have a triangle.');
  write('Enter side 1:  ');
  readln(a);
  write('Enter side 2:  ');
  readln(b);
```

```
  write('Enter side 3:  ');
  readln(c);

  (* Calculations...
   * We first determine which side is the longest, and call it longest.
   * The other two sides will be called leg1 and leg2.
   *)
  if (a > b) and (a > c) then
    begin
      longest := a;
      leg1 := b;
      leg2 := c;
    end
  else if (b > a) and (b > c) then
    begin
      longest := b;
      leg1 := a;
      leg2 := c;
    end
  else
    begin
      longest := c;
      leg1 := a;
      leg2 := b;
    end;

  (* Test for being a real triangle. *)
  if longest < leg1 + leg2 then
    writeln('Yes, this is a triangle.')
  else
    writeln('No, the three sides cannot form a triangle.');
end.
```

```
(* treadmill.pas - Determine how many calories are burned on the treadmill.
 * We need the following numbers from the user:
 * Speed, incline, time, weight
 * Formula comes from
 * https://www.livestrong.com/article/34973-calculate-treadmill-calories
 *)
program treadmill;

var
  pounds, kg, mph, mpm, incline, minutes, oxygen, cpm, calories: real;

begin
  write('Enter your weight in pounds:  ');
  readln(pounds);
```

```
  write('How fast did you go (mph)?  ');
  readln(mph);
  write('What was the % incline?  ');
  readln(incline);
  write('How many minutes did you walk/run?  ');
  readln(minutes);

  (* Calculations:  Start with unit conversions.  Convert speed to meters
   * per minute, percent grade to a decimal, and weight to kilos.
   *)
  mpm := 26.8 * mph;
  incline := incline / 100.0;
  kg := pounds / 2.2;

  (* Now we are ready for the oxygen formula:  ml per kg per minute.
   * It assumes 3.7 mph represents walking and uses a different for running.
   *)
  if mph <= 3.7 then
    oxygen := 0.1 * mpm + 1.8 * mpm * incline + 3.5
  else
    oxygen := 0.2 * mpm + 0.9 * mpm * incline + 3.5;

  (* Now calculate calories per minute, and the total calories. *)
  cpm := oxygen * kg / 200.0;
  calories := cpm * minutes;

  (* Output.  Precede the output with an empty writeln to visually
   * separate the input from output.
   *)
  writeln();
  writeln('Your speed was ', mpm, ' meters per minute.');
  writeln('You consumed ', oxygen, ' ml of oxygen per kg per minute.');
  writeln('You burned ', cpm, ' calories per minute.');
  writeln('Finally, you burned ', calories, ' calories.');
end.
```

Practice using if-statements

1. In Albuquerque, bad weather is often heralded by what is called the East Canyon wind. According to www.theweatherprediction.com/weatherpapers/011/index.html, meteorologists have determined a formula for predicting an East Canyon wind. It goes like this: Take the pressure readings (in millibars) from Albuquerque, Colorado Springs and Amarillo. You should expect an East Canyon wind if the average of the pressures at Colorado Springs and Amarillo exceeds that at Albuquerque by at least 5. How would we encode this algorithm in Pascal?

2. Ask the user for the current time in Greenville. Determine the time in Seattle (3 hour time difference).

3. Ask the user to enter the lengths of 3 sides of a triangle. Determine if this triangle is equilateral (all sides equal), or isosceles (two sides equal) or scalene (no sides equal).

4. Ask the user to enter the 3 sides of a triangle. Determine which side is the longest and call this side "c", and call the other 2 sides "a" and "b". Determine if this is a right triangle by testing if $c^2 = a^2 + b^2$.

5. Given a number of seconds (which could be a large number), output the equivalent time in minutes:seconds, or hours:minutes:seconds as appropriate. For example, if the input is 200, then we would say this is 3 minutes and 20 seconds.

6. Ask the user to enter the date, in the form of month/day. Determine if this is a valid date. For example, 1/45 is invalid. You may assume it is not a leap year.

7. Ask the user for an integer, and determine if it's a 4-digit number or not.

8. A bank would like to devise a scheme for assigning account numbers so that one of the digits, say the last digit, is a "check digit." The purpose is to help prevent someone accidentally referencing an invalid account number. The check digit should be computable from the rest of the digits of the number. And it should not be as trivial as saying "all account numbers must end in 3." Suggest a scheme for computing the check digit, and write an algorithm that will calculate it when given an account number.

9. Given a certain number of cents (1-99), tell the user how to make change. In other words, we want to know how many quarters, dimes, nickels and pennies are needed to make this amount of money. If a certain denomination is not needed, skip it in the output. For example, for 80 cents, only mention the quarters and nickels.

10. There are 2.2 pounds in a kilogram. Design a program that allows the user to convert between one unit and the other. First, ask the user which way to convert. Then, depending on the case, input the pounds or kilograms as appropriate, and output the weight in the other unit.

11. Suppose that a company makes annual contributions to an employee's retirement plan as follows. If you have less than 2 years of service, you receive no contribution. For 2-6 years you receive 5% of your salary. For 7+ years of service you receive 10% of your salary. Design a program that asks the user for an annual salary and how many years of service with the company. The program will then indicate the dollar amount of this year's annual contribution.

12. A person's body mass index is 703 * (weight in pounds) divided by the square of (height in inches). If this exceeds 30, the person may be obese, and if over 25 the person may be overweight. Design a program to implement this formula in Pascal.

# LOOPS

Loops are an extremely useful feature in any programming language. They allow you to direct the computer to execute certain statements more than once. In Pascal, two ways we can perform repetition are called `while` loops and `for` loops. The `while` loop is easier to learn first because it resembles the `if` statement that you already know. However, in the long run, the `for` loop is much more commonly used in real programs.

## Repetition using the `while` statement

"Stir the soup until it boils." The concept of doing something repeatedly is second nature to us. Now let's teach the computer how to repeat an action.

In Pascal, we use a `while` statement to allow some statements to be performed several times. What is really nice is that the format of a while statement is very similar to if:

```
while <condition> do
  (* steps that will be repeated *)
```

This group of statements that starts with the while statement and includes the body is called a *loop*. In particular, we would call this a "while loop" since it began with the word while. There is another type of loop called a "for-loop" that we will see later. Performing the body of the loop each time is called one *iteration* of the loop.

Also, note that the body of a while loop has the same structure as for an if-statement: It may be a single statement, or it may be any number of statements enclosed between "begin" and "end".

Here is how a while loop works: Pascal will first test the condition. If it is already false, then the entire loop is skipped and we go on to the next piece of the program. But if the condition is true, then we enter the loop and perform the statements in the body. When the body is finished, Pascal will re-evaluate the condition. If it is still true, the body is performed again! The process repeats itself. At some point, the condition needs to become false, so that we can eventually exit the loop.

The most basic kind of loop we need to write is how to teach the computer how to count. The following code will print the numbers from 1 to 5.

```
writeln('I''m going to count to 5:');

n := 1;
while n <= 5 do
  begin
    writeln(n);
    n := n + 1;
  end;
```

```
writeln('I''m finished counting.');
```

Many of the while loops we will write will have the same basic structure of this example. And once you have written one loop, it is easy to adapt it to similar other problems. For instance, if we can count to 5, then we can easily modify the above code so that it can count to 1000 instead of 5. How would we make this change?

Another simple modification we can make to the above loop is to count by fives:

```
writeln('Let''s count by fives from 0 to 100.')

n := 0;
while n <= 100 do
  begin
    writeln(n);
    n := n + 5;
  end;
```

Discussion:

1. How would we modify the code so that it also prints out the sum of the numbers that it prints? In other words we want the sum of the multiples of 5 from 0 to 100 inclusive.

2. How would we modify the above code so that it prints the numbers in reverse order, i.e. from 100 down to 0?

3. Try this experiment: What errors result if you forget the word “begin” or the word “end” (or both) around the body of the loop?

4. What numbers are printed by this Pascal code?
```
n := 8;
while n < 12 do
  begin
    writeln(n);
    n := n + 1;
  end;
```

5. What is the output of this program?
```
n := 20;
while n >= 5 do
  begin
    writeln(n);
```

```
        n := n - 1;
      end;
    writeln('After the loop, n equals', n);
```

Important observation: With a loop there is very often some "counter" variable that keeps changing on every iteration. In the above examples, it was this variable n. Notice that before the loop, n received some *initial value*, such as zero. Then, when we introduced the while loop, we established some *limit* on the value of n, such as 100. Finally, at the end of the loop body, we *increment* the counter variable by some amount. Almost always this increment is 1, but in the above example we saw that it could be something else like 5.

So, when you write loops, you should remember: *initial value, limit value, increment*

Since all the while loops we have seen have had the same basic structure, it is useful to keep in mind a basic pattern. Suppose you want a loop to do something exactly 20 times. Here is how you would do it using a while loop:

```
n := 1;
while n <= 20 do
  begin
    (* do whatever you need the loop to do on each iteration *)
    n := n + 1;
  end;
```

The way the above code is written, it is guaranteed to perform the body exactly 20 times. If you wanted to do it 30 times, just change the 20 to 30. As a matter of personal preference, some people like to begin the variable n at zero, and use < in the while condition instead of <=. In other words, the 20 iterations would be numbered 0..19 instead of 1..20.

<u>Application: Prime numbers</u>

A prime number is a positive integer that has exactly 2 factors, namely 1 and itself. Examples of prime numbers are 2, 3, 5, 7, 11, 13, etc.

How can we tell if a number <u>n</u> is prime? Here is an algorithm to do it.

- For each possible <u>divisor</u> from 1 to n, see if it divides evenly into n.
- We need to <u>count</u> how many times we find a divisor that divides evenly into n. Store this in another variable.
- When finished, see if the count is 2.

Let's illustrate the algorithm with some examples.

Is 5 a prime number?

- The possible divisors to test are 1, 2, 3, 4, 5.
- Do they divide evenly into 5? 1 → yes, 2 → no, 3 → no, 4 → no, 5 → yes
- We counted 2 divisors. It is prime.

Is 6 a prime number?

- The possible divisors to test are 1, 2, 3, 4, 5, 6.
- Do they divide evenly into 6? 1 → yes, 2 → yes, 3 → yes, 4 → no, 5 → no, 6 → yes
- We counted 4 divisors. It is NOT prime.

Here is the Pascal program to determine if a number is prime. It follows exactly from the algorithm just described above.

```
program prime;

var
  n, count, divisor : integer;

begin

  writeln('Enter a positive integer:  ');
  readln(n);

  (* The divisor goes from 1 to n, inclusive, just like we are able
     to count from 1 to n.  Also, start count at 0.                *)
  count := 0;
  divisor := 1;
  while divisor <= n do
    begin
      if n mod divisor = 0 then
        count := count + 1;
      divisor := divisor + 1;
    end;

  if count = 2 then
    writeln('Your number is prime.')
  else
    writeln('Your number is not prime.')

end.
```

Discussion:

1. Where is the variable `count` used in this program?

2. Where is the variable `divisor` used in this program?

3. What is the difference between the variables `divisor` and `n`?

4. If the value of `n` is 9, then what are the values of `count` and `divisor` at the end of the program?

More about loops

Earlier we wrote a loop to some simple counting and adding. Now, let's create a more useful loop: let's add five input numbers. We want to allow the user to enter 5 numbers, and calculate their sum. We'll need a loop, and inside the loop we need to do the following:

- Ask the user for the next number.
- Continually add this next number to the sum.
- Keep track of how many numbers so far have been read.

The information that we need to keep track of tells us what variables we want.

*Important observation: If you want to add or count something, make sure your variable starts at zero.*

Here is the program.

```
(* add5.pas - Let's ask the user for 5 numbers, and then find
 * their sum.  It's critical to think about the variables we need.
 *)
program add5;

var
  count, sum, NextNumber : integer;

begin
  count := 0;
  sum := 0;
  while count < 5 do
    begin
      write('Enter a number:  ');
      readln(NextNumber);
      sum := sum + NextNumber;
      count := count + 1;
    end;
  writeln('The sum of your numbers is ', sum);

end.
```

Discussion:

1. How would we modify the above example program so that it reads all 5 numbers on the same line?

2. Maybe the number of values to sum won't be 5 every time. So, how would we modify the above example program so that it first asks the user how many values to sum?

3. How would we modify the above example program so that we don't require the user to tell us in advance how many numbers will be entered? Instead, the user will input a negative value (e.g. –1) to signal that there is no more input. Consider how you would change the while loop's condition. Hint: You will no longer need the variable count.

4. What does this code accomplish?

```
count := 1;
sum := 0;
while count <= 100 do
  begin
    sum := sum + count;
    count := count + 1;
  end;
writeln(sum);
```

5. What is the output of the following code?

```
x := 10;
y := 1;
while y < x do
  begin
    if x mod y = 0 then
      writeln(y);
    y := y + 1;
  end;
```

<u>Common loop mistakes</u>

We all make mistakes. Be patient and forgiving with yourself and others. When composing programs, we need to be careful and methodical, because one small mistake can have a profound effect on the output. There are three easy ways to mess up a loop, and I'll illustrate them here with the task of trying to print the numbers from 1 to 10 inclusive.

In each example, see if you can determine exactly why the code is in error, and how you would fix it.

Mistake #1 – Condition initially false. In this case, the loop will not execute at all. It gets skipped.

```
num := 1;
while num > 10 do
  begin
    writeln(num);
    num := num + 1;
  end;
```

Mistake #2 – Infinite loop. Here, the condition is always true; never has a chance to become false.

```
num := 1;
while num <= 10 do
  begin
    writeln(num);
  end;
```

Mistake #3 – Off by one error. This means we have one iteration too many or too few. I will illustrate both ways of making this error.

```
num := 1;
while num < 10 do
  begin
    writeln(num);
    num := num + 1;
  end;
```

```
num := 1;
while num <= 10 do
  begin
    num := num + 1;
    writeln(num);
  end;
```

<u>Error checking of input</u>

Now that we know how to write loops and if-statements, there is a lot we can do. As you know, very often in our programs, we need the user to enter some input. But what if the input doesn't make any sense? The easiest thing is for the program to just accept the invalid input and ultimately spit out some worthless answer. But it would be more useful if the program can warn the user of a problem with the input, and then give the user another chance. This is called error checking.

Let's say we want the user to enter a positive number. If the user enters zero or a negative value, we need to loop until the user supplies us with a positive number.

The key to error checking is to use a boolean variable. I like to have a variable called `NeedInput`. This variable is initially set to `true`, because we definitely need input from the user. If the user supplies us with good input, then the value of `NeedInput` can be set to `false`, and this will be our signal that we don't need to ask again for the input. But if the input is invalid, we need to print some kind of error message and have the user try again.

Here is the code to do error checking. You will find it very useful in this class and in the future!

```
NeedInput := true;
while NeedInput do
  begin
    write('Enter a positive number:  ');
    readln(value);
    if value > 0 then
      NeedInput := false
    else
      writeln('Sorry, try again.');
  end;
```

Exercise:

1. How would you perform error checking if we wanted the user to enter an odd number? In other words, rewrite the above code to accomplish this, making the appropriate changes, even though the overall structure will be the same.

2. How would you perform error checking if we wanted the user to enter just a single digit from 0 to 9?

<u>The `for` loop</u>

Pascal provides a second way to write a loop, using the keyword `for` instead of `while`. We generally prefer to use a `for` loop when we know in advance how many iterations we need to perform. Here is the format of a `for` loop:

```
for <variable> := <first number> to <last number> do
  begin
    (* steps to perform on each iteration *)
  end;
```

The most common way to write a `for` loop is to count, like this example to print the consecutive integers from 5 to 15, inclusive:

```
for i := 5 to 15 do
  begin
    writeln(i);
  end;
```

Similarly, if we want to print 1-10 using a `for` loop, we would say:

```
for i := 1 to 10 do
  begin
    writeln(i);
  end;
```

It is generally a good practice to use the variable i to be the counter of a `for` loop. The letter i stands for "index".

One convenient feature of the `for` loop is that the Pascal system increments the value of i automatically. We do not include an increment statement for our counter variable. If we had used a while loop instead of a `for` loop, we would have to explicitly include a statement "i := i + 1;" at the end of the body of the loop.

It is also possible to write a `for` loop that counts backwards. But in this case you need to use the word `downto` instead of `to`:

```
for i := 10 downto 1 do
  begin
    writeln(i);
  end;
```

As you might expect, the above loop will print numbers in descending order from 10 to 1.

Here is an interesting technical note. After a for-loop, the variable (e.g. i) that you used to control the loop retains the value that it had on the last iteration. And you can write a little Pascal program to verify this is actually the case. For example, if a for-loop uses the variable i to count from 1 to 20, then after the loop, the value of i is still 20, until it is reassigned a new value or is used in another for-loop.

Discussion:

1. What does this code accomplish?

```
sum := 0;
for i := 2 to 6 do
  begin
    writeln(i);
    sum := sum + i;
  end;
writeln(sum);
```

2. What does this code accomplish?

```
sum := 0;
```

```
for i := 1 to 5 do
  begin
    sum := sum + i;
    writeln(sum);
  end;
```

3. What does this code accomplish?

```
for i := 3 to 7 do
  begin
    writeln(i * i);
  end;
```

4. What does this code accomplish?

```
count := 0;
for i := 1 to 100 do
  begin
    if i mod 7 = 0 then
      count := count + 1;
  end;
writeln(count);
```

5. Write code containing a loop that will print the multiples of 4 from 4 to 48 inclusive.

6. Write code containing a loop that will count how many positive integers are evenly divisible into the number 50.

7. Write code containing a loop that will print all 3-digit numbers that end in 7. When you are done, you might notice that there is a lot of output. So, modify your program so that it prints only 12 numbers per line.

8. Ask the user to enter a digit. Then, print all 2-digit numbers that end with this digit.

9. Rewrite the prime and add5 programs using for-loops instead of while-loops.

10. Let's write a loop that illustrates compound interest. Suppose you have $5000, and you invest it in an account that pays 5% per year. Show how much money you have at the end of each of the next 10 years. Format the figures to 2 decimal places. Also make sure that your formatting allows all of the decimal points to be vertically aligned.

11. Write two loops that produce the following output:

```
   1   2   3   4   5   6   7   8   9  10
   1   4   9  16  25  36  49  64  81 100
```

Each loop should print a row of numbers. The second row shows the squares of the numbers in the first row. You will need to use formatted output to get the rows to line up. Also, don't forget to print a newline after the first loop.

Another method to convert between binary and decimal

Earlier you learned how to convert between binary and decimal notation for positive integers. In this section I will show you a new approach that is much more amenable for writing a computer program. Often in computer science, it is very nice to have multiple ways to solve the same problem, because one approach may be more straightforward than the other, even though both methods may be technically correct. And you may have a personal preference of one method over the other. The approach here relies on using loops instead of having to calculate many powers of 2.

To convert from binary to decimal, follow these steps.

1. We are given a binary string, and we need to convert it to a base-10 value. We're going to start the value at zero. Then, we read the bits one at a time in order to gradually compute the total value of the binary string.
2. For each bit in the binary string from left to right:
    a. Double the current value
    b. Add the value of the bit.

   In a nutshell, this step does the following:
   Since the bit we are reading is either 0 or 1, there are just two possibilities. We either simply double the value, or we double and then add 1.

If the given binary string begins with 0's, you may ignore them. For instance, 00111 is the same as 111.

Let's look closely at an example. Notice that each time we read one bit of the binary number, the value increases. Let's convert the binary string 101110. The initial value is 0.

| Read 1 | Read 0 | Read 1 | Read 1 | Read 1 | Read 0 |
|---|---|---|---|---|---|
| Double the existing 0 and then add 1. | Double the existing 1. | Double the existing 2 and then add 1. | Double the existing 5 and then add 1. | Double the existing 11 and then add 1. | Double the existing 23. |
| 2 (0) + 1 = 1 Value is now 1. | 2 (1) + 0 = 2 Value is now 2. | 2 (2) + 1 = 5 Value is now 5. | 2 (5) + 1 = 11 Value is now 11. | 2 (11) + 1 = 23 Value is now 23. | 2 (23) + 0 = 46 Value is now 46. |

Here are some more example computations. At each step, the bit we are currently reading will be underlined:

| | | |
|---|---|---|
| Let’s convert 1100010. | Let’s convert 10101. | Let’s convert 111000. |
| Value starts at 0. | Value starts at 0. | Value starts at 0. |
| 1100010 → 2 (0) + 1 = 1 | 10101 → 2 (0) + 1 = 1 | 111000 → 2 (0) + 1 = 1 |
| 1100010 → 2 (1) + 1 = 3 | 10101 → 2 (1) + 0 = 2 | 111000 → 2 (1) + 1 = 3 |
| 1100010 → 2 (3) + 0 = 6 | 10101 → 2 (2) + 1 = 5 | 111000 → 2 (3) + 1 = 7 |
| 1100010 → 2 (6) + 0 = 12 | 10101 → 2 (5) + 0 = 10 | 111000 → 2 (7) + 0 = 14 |
| 1100010 → 2 (12) + 0 = 24 | 10101 → 2 (10) + 1 = 21 | 111000 → 2 (14) + 0 = 28 |
| 1100010 → 2 (24) + 1 = 49 | Answer is 21. | 111000 → 2 (28) + 0 = 56 |
| 1100010 → 2 (49) + 1 = 98 | | Answer is 56. |
| Answer is 98. | | |

Now, let’s go the other way. To convert from decimal to binary, follow these steps:

1. We are given a base-10 number that we need to convert to binary. First, write this number in brackets.
2. If the number in brackets is 1, then we are finished. You can now remove the brackets. But if this number in brackets was not 1, then continue with the next step.
3. If the number in brackets is even, then we “take out a 0”. This means you write a 0 immediately to the right of the bracketed number. At the same time, divide the bracketed number by 2. For example, if the bracketed number was [ 30 ], then we now write: [ 15 ] 0.
4. If the number in brackets is odd, then we “take out a 1”. This means you write a 1 immediately to the right of the bracketed number. At the same time, subtract 1 from the bracketed number and then divide by 2. For example, if the bracketed number had been [ 25 ], then we now write [ 12 ] 1.
5. Continue with step 2.

Here are some examples to illustrate the procedure. Notice that as we work, we are slowly building the binary representation from right to left.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| [ 22 ] | | [ 37 ] | | [ 114 ] | | [ 83 ] | |
| [ 11 ] | 0 | [ 18 ] | 1 | [ 57 ] | 0 | [ 41 ] | 1 |
| [ 5 ] | 10 | [ 9 ] | 01 | [ 28 ] | 10 | [ 20 ] | 11 |
| [ 2 ] | 110 | [ 4 ] | 101 | [ 14 ] | 010 | [ 10 ] | 011 |
| [ 1 ] | 0110 | [ 2 ] | 0101 | [ 7 ] | 0010 | [ 5 ] | 0011 |
| | 10110 | [ 1 ] | 00101 | [ 3 ] | 10010 | [ 2 ] | 10011 |
| | | | 100101 | [ 1 ] | 110010 | [ 1 ] | 010011 |
| | | | | | 1110010 | | 1010011 |

Exercise: Write Pascal programs `tobinary.pas` and `todecimal.pas` that perform these conversions. Use the above pseudocode descriptions as a guide. Notice where you will need to write a loop. Test your programs using the examples above.

Nested loop

A nested loop is simply a loop within a loop. These are actually quite common in real computer programs because they allow the computer to do a lot of work with a small number of program statements.

A nested loop is useful whenever we need to process multi-dimensional information. In this course, we will not encounter this very often. But here are some general examples:

- An image consists of rows and columns of pixels.
- A video is essentially a list of images, so this adds a third dimension of time to our images.
- Games often have a 2-D board.
- Airline and hotel reservations. For example, a seat on an airplane is given by a row number and which seat in that row.

In real life, our brains process nested loops all the time! Here are typical examples:

- For each week… for each day of the week… for each hour of the day
- For each hour… for each minute… for each second
- For each building… for each room in the building
- For each book… for each chapter… for each page

I need to caution you that we should not confuse a nested loop with consecutive loops. Just because you see two loops does not automatically mean that one loop must be inside the other. Consider the following examples. In the first case, the two loops are simply consecutive. In the second case, the two loops indeed nest.

```
(* Example #1 *)
count := 0;
for i := 1 to 10 do
  count := count + 1;
for j := 1 to 20 do
  count := count + 1;
writeln(count);
(* ------------------------ *)
(* Example #2 *)
count := 0;
for i := 1 to 10 do
  for j := 1 to 20 do
    count := count + 1;
writeln(count);
```

What is the output of each loop situation here?

As another example, let's suppose we wanted to print a field of star/asterisk (*) characters having 10 rows and 20 stars per row. Here is how it would be done:

```
for i := 1 to 10 do
  begin
    for j := 1 to 20 do
      write('*');
    writeln();
  end;
```

Notice that the nested loop is set up exactly the same way as in Example #2 above. This is because we want 10 outer iterations and 20 inner iterations. We want to print 10*20 = 200 stars. Since we wish to print the stars right next to each other within the line, we use the write statement instead of writeln. If we had used writeln to print each star, this would have printed a newline character after every star.

Also notice the writeln statement in the code. It is positioned immediately after the inner loop, but it is located within the outer loop. This makes sure that we print the newline after each row of stars.

Discussion: Let's reformat the code of the previous example and experiment with it. Line numbers have been included for convenience.

```
1   for i := 1 to 10 do
2    begin
3      for j := 1 to 20 do
4        begin
5          write('*');
6
7        end;
8
9      writeln();
10   end;
11
```

1. What is the output of the code as it is written?

2. How would the output appear if we remove the `writeln` statement on line 9?

3. How would the output appear if we move the `writeln` statement on line 9 to line 6?

4. How would the output appear if we move the `writeln` statement on line 9 to line 11?

5. How would the output appear if we move the `write` statement on line 5 to line 8, and we remove the `writeln` statement on line 9?

6. How would the output appear if we move the `write` statement on line 5 to line 11, and we remove the `writeln` statement on line 9?

Algorithms with loops

Example #1: The Euclidean algorithm. Given two numbers, it finds their greatest common divisor.

Algorithm:

1. Let m = the larger number, and let n = the smaller number.
2. Let r = the remainder after dividing m/n.
3. If r = 0, then our answer is n, and we are done. But if r ≠ 0, let m = n, n = r, and go to step 2.

Let's try out this algorithm. Do you understand the steps? Will the procedure work? The algorithm here is rather tersely stated. It said nothing about I/O, and if we ever wanted to write this into a computer program, we would have to fill in those details.

Exercise: Work out the Euclidean algorithm above with the numbers 32 and 84. Does the algorithm give the correct answer?

Example #2: Allow the user to play a simple guessing game.

For example, we could have the user guess a number between 1 and 100. We need to set the maximum number of guesses to be $\log_2 100$, which is about 7. The secret number can be hard-coded in the program, or better yet, we can use a built-in function to select a random number. However, it may be easier to test our algorithm if we hard code the correct answer.

Algorithm:

1. Let the secret number be 42.
2. Let the maximum number of allowed guesses to be 7.
3. Initialize the number of actual guesses accrued to be 0.
4. While the number of guesses < maximum guesses, loop:
    a. Ask the user for a guess
    b. Increment the number of guesses made

c. If the guess matches the secret number:
   Win! Congratulate the user.
   Break from the loop, as the game is over.
d. Otherwise, at this point, we know the guess is wrong.
   If the guess > secret number,
   Tell the user the guess was too high.
   Otherwise,
   Tell the user the guess was too low.

5. If we get this far, that means the user has run out of guesses and has lost the game.

Discussion:

How does the above algorithm determine that the player has won or lost the game?

Practice with loops

1. Write a for-loop that will print the positive integers 1-25 inclusive.

2. The Fibonacci sequence is this list of numbers: 1, 1, 2, 3, 5, 8, 13, 21, etc. The first two numbers are 1 and 1, and each subsequent number is the sum of the previous two. Using a loop, initialize an array to contain the first 20 Fibonacci numbers.

3. Output the first twenty powers of 2. (2, 4, 8, 16, … , $2^{20}$) Also find the sum of these numbers.

4. Ask the user to enter a number. Print all the divisors of this number. For example, if the user chooses 10, then print out the numbers 1, 2, 5 and 10. Also tell the user how many divisors.

5. Given a positive integer, how would we use a loop to
   a. figure out how many digits it has?
   b. Given a positive integer, how would we find the sum of the digits?

STRINGS

Objectives:

- How text data is internally represented as a string
- Accessing individual characters by an index
- Operations on strings: concatenation, comparison
- Using Pascal's built-in string functions

Not all data is numerical. The purpose of the string data type is for us to keep track of text information. A string is a collection of any number of characters. And a character is basically anything that you can type on the keyboard, including letters, digits, punctuation symbols, and blank spaces, tabs and newlines. (A newline is a special character that results when you hit the Enter/Return key and signals your computer that it should go to the next line of a text document.)

If you know that your string will contain exactly one character, then you could alternatively use the `char` data type. But for most purposes this will not be necessary.

In Pascal, we denote a string constant by enclosing it in either single quotes 'like this'. Pascal allows us to declare variables to hold string values, just like any other type (boolean, integer, real). So, if we declare `name` to be a string, the following assignment statements are valid:

```
name := 'Nick Charles';
name := 'N';
name := '';
```

In the last example above, there was nothing between the two single-quotation marks. This is called the *empty string*, not to be confused with strings containing only invisible characters such as `' '` or `'      '`.

Accessing a character

Since a string is a collection of characters, it makes sense to want to access individual characters or parts of the string. Pascal uses the brackets [ ] to access portions of a string. Before I show you all the syntax for using the brackets, it is important to understand how the individual characters of a string are numbered.

Inside a string, each character lives inside a cell, and these cells are numbered sequentially from one. Thus, if s is a string variable and contains the word "horse", then the individual characters are located as follows:

| Index | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Character | 'h' | 'o' | 'r' | 's' | 'e' |

To access an individual character of a string, simply put its index inside brackets after the name of the string. So, s[1] refers to the first character of the string variable s, which in this case is 'h'. Similarly, s[2] is 'o', s[3] is 'r', s[4] is 's' and s[5] is 'e'. Right away, you should notice something about the size of the string and its index values. In this example, our string has 5 characters, and they are located at indices 1..5 (i.e. 1 through 5 inclusive). No matter how long a string is, its first character is always at index 1. The index of the last character is always equal to the length of the string.

Think of an index like an apartment number. The letter 'r' is located at index 3.

*Concatenation* is another fundamental operation on strings. It means to merge two strings into each other. We use the + sign. It works for string variables, string constants, or any combination. Example: 'ab' + 'cde' results in the string 'abcde'.

Discussion:

1. In the string s = 'abcdefghijklmnopqrstuvwxyz', which letters are located at s[4] and s[24]?

2. If s = 'Salt Lake City', then what are the indices of the letters in the second word "Lake"?

3. What is the output of the following code?
```
s := 'dolphin';
writeln (s[1] + s[4]);
writeln (s[4] + s[1]);
t := '';
for i := 1 to 7 do
  t := s[i] + t;
writeln(t);
```

4. Let the strings s and t be 'cat' and 'house', respectively. Create an expression using s and t that equals 'house cat'.

<u>Changing a string</u>

Modifying strings in Pascal is straightforward. Once you have a variable that contains a string, you are free to reassign it to a new string, or you can tell Pascal that you just want to change one of its characters, such as the $5^{th}$ character.

For example, suppose the string variable s denotes 'doghouse'. To change one letter of a string, all we need to do is write an assignment statement, specifying the string, which position we want to change, and the new character we want in that position. In this case, we say we want the first letter in the variable s to become the capital letter D. Therefore, the resulting assignment statement becomes:

```
s[1] := 'D';
```

Here is another example. Suppose s is a string containing several words, and we want to modify s by appending a period at the end of the string. Here is how to do it:

```
s := s + '.';
```

As another example, this code:

```
word := 'dog';
word := word + 's';
```

results in the word 'dogs' being stored in the variable called `word`.

## Comparing strings

You have just seen concatenation of strings, but there is more.

We can compare two strings using relational operators, and thereby compare them just like we compare numbers. All six relational operators (=, <>, <, <=, > and >=) can be used on strings. Pascal relies on ASCII values of each character to determine alphabetical order. For instance, 'a' is considered the "lowest" letter of the alphabet, and 'z' is the "highest" letter. So, 'a' < 'b', 'b' < 'c', and so on. Another way to think of < is like this: if s1 and s2 are strings, then s1 < s2 means that s1 should appear earlier in the dictionary than s2. For example, 'cat' is less than 'dog'.

Long ago, someone made the arbitrary decision that capital letters are "less than" lowercase letters. Consequently, 'Cat' < 'cat'. Also, the absence of a character is less than any real character. For example, 'dog' is less than any 4-character string that begins with 'dog'. In particular, 'dog' < 'dogs'.

Example: Here is a program that calculates useful data about a string. Note the use of an if-statement inside the loop. We use comparison operators to see if a character is within some range, e.g. of lowercase letters. Also note that the for-loop does not need a begin/end since the body contains only one statement.

```
(* stringcount.pas - Read in a string, and then count how many
 * of its characters are letters, digits, and spaces.
 *)
program stringcount;

var
   s : string;
   i, NumSpaces, NumLowercase, NumCapital, NumDigits : integer;

begin
```

```
  write('Please enter a string:  ');
  readln(s);

  (* We can count everything with one loop. *)
  NumSpaces := 0;
  NumDigits := 0;
  NumCapital := 0;
  NumLowercase := 0;

  for i := 1 to length(s) do
    if s[i] = ' ' then
      NumSpaces := NumSpaces + 1
    else if (s[i] >= '0') and (s[i] <= '9') then
      NumDigits := NumDigits + 1
    else if (s[i] >= 'A') and (s[i] <= 'Z') then
      NumCapital := NumCapital + 1
    else if (s[i] >= 'a') and (s[i] <= 'z') then
      NumLowercase := NumLowercase + 1;

  writeln('Your string has');
  writeln(NumSpaces, ' spaces');
  writeln(numDigits, ' digits');
  writeln(NumCapital, ' capital letters and');
  writeln(NumLowercase, ' lowercase letters.');
end.
```

Discussion: What enhancements to this program would you suggest?

## Important string functions

Functions are written by people to save us time and effort in coding. Our Pascal installation has several useful built-in functions for strings. Here are some of the most commonly used string functions.

| Function | Meaning | Example |
|---|---|---|
| length(s) | How many characters in s | length('dog') returns 3 |
| pos(letter, s) OR pos(substring, s) | Returns index of first occurrence of the letter or pattern. Returns 0 if not found. | s := 'dog'; pos('o', s) returns 2 |
| upcase(s) | Returns s with all of its letters converted to uppercase. Does not change s. | s := 'dog'; upcase(s) returns 'DOG' |
| lowercase(s) | Analogously returns s with all of its letters converted to lowercase. Does not change s. | s := 'DOG'; lowercase(s) returns 'dog' |
| copy(s, i, count) | Returns a portion of the string s. The substring being returned starts at s[i] and contains count characters. | s := 'friendly'; t := copy(s, 4, 3); will set t to the string 'end' |

For more information, you can find detailed documentation on string functions here:
https://www.freepascal.org/docs-html/rtl/system/stringfunctions.html

Cryptography

As you know, a string is a series of characters. It turns out that every character that can be typed or printed out has an internal numerical representation. This representation is called ASCII code. For example, the capital letter 'A' is represented inside the computer by the number 65. Since character data is essentially numerical data, we can perform mathematical functions on our characters. This capability is exploited most famously in the world of cryptography.

Pascal has two useful character functions to support cryptography.

- `ord`(character) → returns the numerical ASCII code of this character
- `chr`(ASCII code) → returns the character having this ASCII code

For example, ord('B') returns 66, and chr(66) returns 'B'.

Here is a simple example illustrating how cryptography works. Let's suppose we wish to encrypt a string by adding the number 3 to all of its characters. The pseudocode would work as follows:

```
for each letter in the string
    value := ord(letter);
    print out the encrypted letter chr(value + 3)
```

Exercise: Suppose `s` is a string. How would we determine the following?

1. If it contains the letter T, capital or lowercase
2. The number of times the capital letter T appears
3. The location of the first capital T
4. The location of the second capital T, assuming that it exists
5. If it contains a digit
6. If it contains at least 2 vowels
7. The number of vowels
8. The locations of all the vowels
9. If it has more than 5 characters
10. The fifth character
11. The last 3 characters
12. If the first and last characters are the same
13. If all of its letters are capitalized

More questions about strings:

14. How would you reverse a string?

15. Suppose `pos('g', s)` returns 3. What can we conclude about the string s?

16. What is the output of the following code?
```
word1 := 'well';
word2 := 'done';
word1 := word1 + word2;
writeln (word1);
```

17. What is the string contained in s2 when the following code finishes?
```
s1 := 'dolphin';
s2 := '';
for i := 1 to length(s1) do
  if pos(s1[i], 'aeiou') > 0 then
    s2 := s2 + s1[i];
```

18. Repeat the previous question, but this time, suppose the last line reads:
```
      s2 := s1[i] + s2;
```

19. What is the output of the following code?
```
s1 := 'dolphin';
s2 := '';
s3 := '';
for i := 1 to length(s1) do
  if pos(s1[i], 'aeiou') > 0 then
    s2 := s2 + s1[i];
  else
    s3 := s3 + s1[i];
writeln(s2);
writeln(s3);
```

20. Fill in the blanks in the following code so that s2 contains the word “Lake”.
```
s1 := 'Salt Lake City';
s2 := copy(s1, ________, ________);
```

21. What string is contained in s4 after the following code executes?
```
s1 := '1';
s2 := s1 + '2' + s1;
s3 := s2 + '3' + s2;
s4 := s3 + '4' + s3;
```

22. Design a program that asks the user for a string, and then prints out every fiftieth character of the string. In other words, the characters at index 50, 100, 150, 200, etc.

23. Ask the user for a word. Determine if the word contains a double letter. A double letter means that you have 2 consecutive characters that are the same.

24. Suppose s is a string variable containing a time, such as ‘6:17’. There is an integer on either side of a colon. Assume this represents minutes and seconds. Output the total number of seconds this represents. For example, in this case 6:17 equals 377 seconds.

25. A computer system has a rule about user names. They must be 8 characters long, and must contain at least 2 letters and at least 2 digits. Given a proposed user name, test if it’s valid.

26. Ask the user to enter a string. Determine if it is a valid identifier in Pascal. In other words, the first character is a letter or underscore, and each other character is a letter, underscore or a digit.

27. Suppose s is a string variable. How would you capitalize all of the vowels in this string?

28. Suppose s is a string containing a first and last name. Print out this name, with the last name followed by the first name. For example, if s = “Al Capone” you should print “Capone Al”.

29. Suppose S and T are string variables. How can we determine if S and T have any characters in common? For example, S and T might contain words, and we are interested if these two words have any letters in common.

30. Suppose s is a string that contains lowercase letters. The letters may be p, n, d or q. Each letter represents a coin in a piggy bank. Read the characters of this string to determine the total amount of dollars represented. The coins may appear in any order. For example, if s = “qdppnq”, this adds up to 25 + 10 + 1 + 1 + 5 + 25 = 67 cents or 0.67 dollars.

# ARRAYS

After integer, real, boolean and string, we are now ready to learn a fifth data type: the array.

The major benefit of the array is that it allows us to store many values and associate them with the same variable. For instance, suppose your program needed to keep track of 30 numbers. Does it make sense to store them in 30 different variables? We would have to come up with distinct names for these 30 variables. Instead, we can put all these numbers into a single array. The magic of the array is that the basic representation and syntax is like a string.

An array is special – it is a collection of data. It allows you to have many of another type. This means that you can have an array of integers, an array of reals, an array of strings, etc.

Whenever working with arrays, there are 3 basic steps we need to follow:

1. Declare the array. You must first tell Pascal how many cells it should have, and what is the base type of the value inside each cell. If you are ever unsure how big the array should be, it is okay to over-budget. This is because during program execution, the array's size is fixed. It cannot grow or shrink. For example, if you anticipate that an array will contain between 50 and 100 real values, then declare it to have 100. It is always acceptable to leave some cells unused.

2. Initialize the array. It turns out that you can put the values in the array at the same time you declare the array. But this requires that you know what values to put in when you are typing your program. Alternatively, you can put the values into the array later in the program.

3. Use the array. Once the array contains its values, you are free to manipulate them.

## Declaring and initializing

Here is an example. Let's declare a variable called data, and store the numbers 5, 4, 7 and 3 in it as an array like this:

```
var
  data : array[1..4] of integer = ( 5, 4, 7, 3 );
```

There are several items to note in this declaration. First, when you declare an array, you need to indicate the number of elements. The notation "1..4" is called a subrange and denotes the locations of all the values in the array. It means "I want an array, and the individual cells will be numbered 1 through 4." The next thing to notice is the base type which appears after the word "of." Here we have an "array of integer", meaning that the individual cell values are integers. The cells of an array must all have the same base type.

Finally, you see the four initialized values, 5, 4, 7 and 3 being put into the array. But notice that we use an '=' sign and not the ':=' operator. Technically, an array declaration and initialization is not an assignment statement. We declare arrays in the `var` section of the program, just like all the other variables. In the main program, after the word `begin`, you are free to change any of the array elements.

Note that it is possible to initialize an entire array like this only at the point it is being declared. In other words, in the main program, you cannot write an assignment statement using := to assign all the values of an array. The := assignment operator can only assign to one array cell at a time.

After having initialized our array, we can refer to the individual numbers just like we did for strings.

data[1] holds the number 5
data[2] holds the number 4
data[3] holds the number 7
data[4] holds the number 3

Later in the program, you can treat any of these cells like any other integer in your program.

Often when we declare an array, we don't know what values to put in it yet. So, if we just wanted to declare the array, but not initialize it right away, we could simply have done this:

```
var
  data : array[1..4] of integer;
```

It is an error to attempt to access a cell that is outside the range that we declared. In the above example, there is no such location as data[5]. Therefore, when declaring an array, it is important to plan in advance how many cells you want. When in doubt, it is better to create an array that is too big than too small.

Usually, we number the cells of an array starting at 1. But you can use other subranges as you see fit. The only requirement is that the range of indices must be consecutive integers. For example, let's say that you wanted an array to keep track of the number of US astronauts who entered space during the Space Race that culminated with the Apollo program. It would make sense to use the year as an index, and we would want to use the years 1961 through 1972. In this case, when declaring the array, you would specify 1961 and 1972 as the endpoint indices of your array. The declaration would look like this:

```
  astronaut : array[1961..1972] of integer;
```

Exercise:

1. Let's practice writing array declarations. Declare an array called A and arrange for it to contain 10 integers.

2. Declare an array called B and arrange for it to contain 25 strings.

3. Declare an array called C and arrange for it to contain 4 real numbers, and while declaring C, initialize all of the elements to the number 2.5.

We don't usually know the contents of the array when we write a program. A more common scenario is for us to initialize an array based on user input. Let's create an array of 5 integers and ask the user what values should be inserted. First, we would write the basic declaration like this:

```
var
  a : array[1..5] of integer;
```

If you declare an array but do not give it initial values, then those values are undefined. So, during the main program, we can have the user provide the initial values. We accomplish this using a loop:

```
  for i := 1 to 5 do
    begin
      write('Enter the value for a[', i, ']:  ');
      readln(a[i]);
    end;
```

The length() function works for arrays just as it does for strings. This is handy because later you might decide to change the size of the array, e.g. to 10. You would only need to change the 5 to 10 in the declaration, and not also in the loop(s) that traverse the array. So, the above loop could have started as:

```
  for i := 1 to length(a) do
```

<u>Things to do with an array</u>

So far, you have seen an array declaration and initialization. All that is left for us now is to actually use the array. A very simple task we can perform is to print out all of the array elements. And while we are at it, we can also find the sum of these elements. And we will use a loop.

```
  sum := 0;
  for i := 1 to 5 do
    begin
      writeln('a[', i, '] has the value ', a[i]);
      sum := sum + a[i];
    end;
  writeln('Sum of array values is ', sum);
```

Loops and arrays go very well together. An array has several elements, and often we want to perform the same operation on these elements (print them, add them up, search for a value, etc.). A loop is the perfect way to arrange for that to happen. If your program has an array, then you will almost certainly have loops as well in order to visit all of the cells in the array.

A loop is necessary if you want to print out the contents of an array. In other words, this statement:

```
writeln(a);
```

is not valid. Unfortunately, you cannot read or write entire arrays at once. The I/O procedures read, readln, write and writeln can only work on single values rather than arrays.

## Searching an array

How do you search for something? Suppose a friend of yours just moved into a new house. You have been invited over for dinner. It's a rainy day, so you take your umbrella with you. Your friend is eager to show off all the rooms of the house. Later, after dinner, when it's time for you to leave, you realize that you have misplaced your umbrella. You must have left it in one of the rooms in the house. You begin a systematic search, checking every room… until you find it. Once you have located your umbrella, your search is over. You don't actually need to search the entire house, unless you are unlucky! This real-life searching motif is useful for visualizing how we search an array.

On the other hand, let's say today is laundry day. Each room has its own basket of clothes and linens to wash. In this case, the search for laundry items does require a tour of the entire house. You are not interested in finding just one item to wash. You want all the laundry.

Suppose we have an array of 100 integers, already initialized. And we would like to find the location of a certain target number inside this array. The target could be 42. We'd like to know where this number is. This is an example of a "search" operation. To search means we traverse the entire array from one end to the other, visiting all of the cells. If we come upon a cell containing what we seek, we can immediately make a report each time. By this description, you can tell that we will need a loop, and inside the loop we'll need an if-statement. Here is how to do it in Pascal. Let's assume that "a" is the name of the array. The most important code to accomplish the search is in boldface.

```
(* search version 1 *)
target := 42;
for i := 1 to 100 do
  if a[i] = target then
    writeln('I found the value ', target, 'at location ', i);
```

It turns out that searching an array is a very common operation. And the above code can be adapted to other similar searches. For example, if we wanted to search for the number 16 instead of 42, we would simply change the value of target to 16. No other change to the code would be needed.

There are two important search features to consider:

- What should we do if the target doesn't exist in the array?
- What if we only want to know the first location, not all the locations of the target?

Suppose the target number doesn't exist in the array. Then, the above code will not print anything. This lack of output would be disconcerting to the user. It would be nice if the code could explicitly report that the target value was not found. Therefore, we need to include a new boolean variable called "found" that will keep track of whether the search was successful. In the revised code below, the new statements appear in boldface:

```
(* search version 2 *)
target := 42;
found := false;
for i := 1 to 100 do
  if a[i] = target then
    begin
      found := true;
      writeln('I found the value ', target, 'at location ', i);
    end;

if not found then
  writeln('I was unable to find the value ', target);
```

Note the if-statement after the loop. Once the loop is finished, we can determine if the value of found is still false. It is essential that this if-statement appear after the loop, and not inside the loop. This is because we do not know if the search was futile until we searched everywhere for the target number.

Here is one more detail about the final if-statement. Notice the use of the operator "not". Alternatively, we could have said

```
if found = false then
```

Now, let's consider the second enhancement: What if we only care if the target number exists at all in the array? We are not interested in all of the occurrences; just one will suffice. In this case, we can call off the search once we found the target number. We can use the "found" variable for this purpose. On each iteration of the loop, we can check to see if the value of found has been set to true. If it has, then there no need to keep searching.

But in this case note that because we do not necessarily need to search all 100 cells, we should not use a for-loop. For-loops are generally used when we know the number of iterations in advance. But we don't know when the search will succeed. It's not wrong to use a for-loop here, but it would be inefficient. Why continue searching for something once you found it? Instead, we use a while loop. Therefore, the code becomes:

```
(* search version 3 *)
target := 42;
found := false;
i := 1;
while i <= 100 and not found do
  begin
    if a[i] = target then
      begin
        found := true;
        writeln('I found the value ', target, 'at location ', i);
      end;
    i := i + 1;
  end;

if not found then
  writeln('I was unable to find the value ', target);
```

Study the while condition. We continue as long as there are still cells in the array to search, and we have not yet found the target number.

Discussion:

1. Consider the search version #3 shown above. Suppose 42 is the first number in the array. Explain how the code avoids performing the second iteration of the loop.

2. How would we modify the above code (search version #1) so that it…
   a. Finds all of the odd numbers contained in the array?
   b. Finds the sum of all the odd numbers in the array?
   c. Finds all of the positive numbers contained in the array?

3. In general, how would you find the identity and location of the largest number in an array?

4. Suppose A is an array of integers, and we want to determine if a certain target value exists anywhere in A. Why will the following approach not work? How would you fix it?

```
found := false;
for i := 1 to length(A) do
  if a[i] = target then
    found := true;
  else
    found := false;
```

Sorting an array

Sometimes it's desirable to "sort" an array. This means to rearrange the values of the array so that they are either in ascending or descending order. There are many possible ways to sort an array, but an easy approach is this:

"For each pair of values in the array, if they are out of order, then swap them."

And this idea can be implemented as follows. Let's assume we have an array of 100 integers and we want to sort it ascendingly. We only need to look at each pair of cells once. In the following code, the variables i and j are the indices of two array elements we are examining. Imagine that "I" is our left hand and "j" is our right hand. The first pair of values we need to consider is a[1] and a[2]. The last possible pair would be a[99] versus a[100].

```
for i := 1 to 99 do
  for j := i+1 to 100 to
    if a[i] > a[i] then
      begin
        temp := a[i];
        a[i] := a[j];
        a[j] := temp;
      end;
```

Notice that inside the nested loop, we have an if-statement. This tests to see if two given numbers in the array are "out of order." If we want the array to be sorted ascendingly, then out of order means the first number > the second number. If we had wanted the array sorted descendingly, we simply need to change > to <.

Finally, note that in order to swap the values in two locations, three statements are required. During the swapping process, we need a temporary variable so that we don't lose what we are swapping.

In further computer science courses, you will see other methods of sorting an array that are more sophisticated and efficient.

Arrays of strings

It's important to remember that the individual elements of an array do not have to be numbers. They can be anything, including strings. For example, the following code creates an array of strings and prints each string out, one per line.

```
program food;

var
  a : array[1..3] of string = ('hamburger', 'hot dog', 'pizza');
```

```
  i : integer;

begin
  for i := 1 to 3 do
    writeln(a[i]);
end.
```

Discussion: How can we enhance the above program so that it counts how many times the letter 'g' appears in the array of strings?

Exercise: Now, let's review what we know about arrays.

Assume that A is an array of integers. Explain or show how to find …

1. The last value
2. The largest value
3. The second largest value
4. The smallest value
5. How many values are positive
6. The sum of just the positive values
7. The median
8. The location of the first zero
9. The location of the second zero
10. The location of the last zero

11. Traverse the array to see if it contains any values divisible by 3. If so, print out all such values, along with their index locations in the array. Also find the sum of these values.

12. Determine if this array is sorted in ascending order.

13. Find the smallest positive number in the array, as well as its location.

14. Determine how many distinct values the array contains. For example, the array of numbers ( 1, 7, 4, 1, 4 ) has three distinct values.

15. Determine how many unique values the array contains. In other words, how many values appear only once. For example, ( 1, 7, 4, 1, 4 ) has one unique value, namely the 7.

16. Determine if two consecutive numbers in the array are equal.

17. Determine if any two numbers in the array are equal.

Algorithmic questions using arrays… Design pseudocode solutions to the following problems.

18. Ask the user to enter a sequence of numbers. Store these numbers in an array. Stop reading input once you encounter the same number entered twice in a row.

19. Ask the user to enter a sequence of numbers. Store these numbers in an array. Stop reading input once you encounter a number that has already been entered.

20. From a deck of cards, deal yourself 7 cards. Determine how many of these cards are face cards (Jack, Queen, King).

21. Given a poker hand, determine if it's a full house.

22. Suppose you have a large array. How would you print out
    a. only its first 10 elements?
    b. only its last 10 elements?
    c. every other element (1st, 3rd, 5th, …)?

23. Suppose `temp` is a list of real numbers representing the average temperatures for the 12 months of the year, January through December. Show how we can calculate the average temperature of just the 3 summer months, which are June, July and August.

24. (Challenging problem) I would like to schedule a meeting in Room 106 in Riley Hall. Scan the current course schedule. Find out when the room is being used for a class.

25. (Challenging problem) Suppose s is a string containing numbers separated by commas. For example, s might be the string "5,12,8,3,50".
    a. Create an array called CommaLocation that contains the indices of all of the commas in s.
    b. Next, create an array A that consists of the numbers inside s that are separated by the commas. Use the values in CommaLocation to help you find the numbers.
    c. Print the sum of the numbers in A.

<u>Recap</u>

Now that you have seen strings and arrays, it is helpful to notice that often we use loops with them. And with practice we also notice that there are some common motifs when writing a loop. Take counting for example. We can ask the user to enter several numbers and <u>count</u> how many are even. Similarly, we can scan the elements of an array and count how many of its values are even. And for a string, we can visit each of the characters to count how many are vowels.

It would be worth your time to stop here for a moment and write those three programs, and look at them side by side to see how similar in structure they are.

Other common loop motifs are finding the sum of values, or searching for a particular value. The essential procedure is the same whether you are looking at an array, or a string, or interactive input. For example, searching an array for the value 65 is similar to searching a string for the letter A.

If you take further courses in computer science, you will learn about additional ways to aggregate data, such as multi-dimensional arrays and dynamic lists. These also can be processed using loops.

FILE INPUT and OUTPUT

Ultimately, one goal of computer programming is to write interesting and useful programs, and then test them on realistic input data. For example, counting all the words in a book, or finding all the words that have exactly 9 letters in them, or the words that have 3 vowels, etc. "Interesting" programs tend to be those that have large amounts of input and/or output.

<u>File Output</u>

If you have a program that needs to print out a lot of output, it's a good idea to instead write all the output to a file so that you can view it later. By "a lot" of output, I would say more output than you can easily see on a screen. When we perform file output, we actually have our computer program create a new file to contain the output. It turns out that a `textfile` is its own data type, meaning that we can create a variable in a Pascal program to refer to an actual file on the hard drive. So, now he have learned six data types: integer, real, boolean, string, array, textfile.

Here are the steps to accomplishing file output, followed by an example program.

1. First, we need to make decisions about some names.
    a. The output file needs to have a name on your hard drive or other storage device. Since it will be a text file, it should have a name like `output.txt` or something more specific like `football.txt` or `stocks.txt`.
    b. Within your program, you should have a string variable to hold the name of the output file. A good variable name would be `FileName` or `OutputFileName`. Occasionally, it's convenient to keep the file name as a constant instead of a variable. But this only makes sense if you know the name of the file in advance. If you want the user to have the flexibility of specifying the name of the program when it's running, then it needs to be a variable, not a constant.
    c. Your program also needs a second variable to refer to the file itself. A good name for this variable would be `file` or `OutputFile`.

2. Create the output file by using the built-in `assign`( ) and `rewrite`() procedures. You generally should do this early in the program. For example:

```
assign(OutputFile, OutputFileName);
rewrite(OutputFile);
```

   The first argument is the file, as referenced by this variable inside your Pascal program. The second argument is the name of the file that you want to write to.

   The purpose of calling the `rewrite`() procedure is to tell Pascal that you are ready to erase anything that might already be in the file, so that you can begin writing to it at the beginning.

3. When you are ready to print something to the file, use write() or writeln() as usual. You have seen these procedures used to print output to the screen. To tell them that you instead want to write to a file, you include the name of the file variable as the first argument. As an example:

   ```
   writeln(OutputFile, 'a modern major general');
   ```

   In case you have lots of output, then you probably will write a loop, and inside the loop include one or more writeln statements.

4. At the end of the program, tell Pascal that you are done writing to the file by closing it. To close the file, using the built-in close( ) procedure:

   ```
   close(OutputFile);
   ```

   By closing the file, we ensure that all of the output actually gets written to the file. Because of the way that the operating system usually handles files, if we forget to "close" the file, the last part of our output may get omitted from the file.

```
(* FileOutput.pas - Experiment with file output.  *)
program FileOutput;

var
  OutputFileName : string;
  OutputFile : textfile;

begin
  OutputFileName := 'output.txt';
  assign(OutputFile, OutputFileName);
  rewrite(OutputFile);

  writeln(OutputFile, 'This is a test.');
  writeln(OutputFile, 2.5);
  writeln(OutputFile, 4, 4, 4, 4);
  writeln(OutputFile, 5.0 : 10 : 2);

  close(OutputFile);
end.
```

Exercise:

1. In the above example program, where is the output written? What does the output look like?

2. How would we change the program so that everything that gets written to the file also gets printed to the screen?

3. Would the program still work if we declared the OutputFileName as a constant instead of a variable? Try it and see!

4. How would we change the program so that we first ask the user to provide us with the desired name of the output file?

5. Write a program that prints the numbers 1-100 on the screen, one number per line.

6. Write a second program that prints the numbers 1-100 to a file. Call the file counting.txt.

File Input

If your program needs a lot of input (e.g. adding 10 numbers), then rather than having the user type so much input each time the program is run, it is more convenient to use file input. The steps for accomplishing file input are analogous to what you saw for file output.

1. Decide on the names of the input file name on your hard drive, as well as the file and file-name variables in your program. Conventional choices are: input.txt for the name of the file in your file system, InputFileName as the variable in your Pascal program to contain the file name, and finally InputFile to refer to the file inside the program. Early in the program, it is usually necessary to ask the user to enter the name of the input file. We don't store the file name as a constant unless we are sure that we are going to read the same file every time we run the program… this does not sound too interesting.

2. Prepare the input file for reading. This is a two-step process. First, we call the `assign`() procedure just like we did in the case of file output. However, the second step is different. Instead of calling `rewrite`(), we call `reset`(). It should make sense to you that if we are not writing to the file, then we should not be calling `rewrite`(). Be careful that you do not call `rewrite`(), because you will erase your input file!
   ```
   assign(InputFile, InputFileName);
   reset(InputFile);
   ```

3. When you are ready to read a line of text from your input file, you call readln(), similar to reading interactive input. The only difference is that the first argument is the file variable. For example, if line is a string variable to hold a line of text from the file, we can say:
   ```
   readln(InputFile, line);
   ```

4. It's likely that you need a loop to read the input file. In fact, you probably don't know in advance how many lines the input file has. So, in many cases you need a while loop. And you want to read all the lines in the file. The while loop needs to read lines one at a time as long as we are not done reading the file. The while loop would be introduced this way:
   ```
   while not eof(InputFile) do
   ```

   In Pascal, "eof" is an abbreviation for "end of file". So, this loop is saying "while not end-of-file do…"

5. Often, the line we just read contains an integer value. To convert a string into an integer, we use the built-in StrToInt function. So, if n is an integer, we could say

```
n := StrToInt(line);
```

However, the StrToInt function belongs in the "system utilities" run-time library of built-in functions, which is not loaded by default when we run Pascal programs. To make StrToInt work, at the top of your program, immediately after the program statement, you will need to say:

```
uses sysutils;
```

In addition to StrToInt, there is another function called StrToFloat, which we use if we want to interpret the input as a real instead of integer.

6. When done reading input, close the file. This step is analogous to file output.

```
close(InputFile);
```

So, now that you have seen both file input and file output, you should notice that the structure is similar. The essential steps boil down to these three: opening the file, reading or writing the file, and then closing the file.

You can find more information about Pascal's file I/O here:

- https://www.freepascal.org/docs-html/rtl/system/filefunctions.html
- https://wiki.freepascal.org/File_Handling_In_Pascal

Exercise:

1. Create a text file called input.txt, and enter 10 numbers, one per line. This will be our test input.

2. Based on the steps listed above, write a program that reads input.txt and finds the sum of all the numbers in the input file. Your program should be written in such a way that it does not matter how long the input file is. In other words, don't tell your program to read only the first 10 numbers in the input file. Because we don't need to store the input values as we read them, you do not need an array in your program. Hint: You will need the following variables in the program: InFileName, InFile, line, n, sum. Think about what types these variables are.

3. How would you find the average of the numbers in input.txt?

4. How can we determine the number of lines in an input file?

FUNCTIONS and PROCEDURES

Objectives:

- Why functions (and procedures) exist
- What happens when we enter and leave a function
- Parameters and return values
- How to create and use a function

As you know, a computer program is a list of instructions for the computer to perform in order to solve some problem. But, what happens if we have “a lot” of instructions? It’s just like what happens when any business or group of people has a lot of work to do – they get organized and delegate the work.

A restaurant kitchen is a busy place. Some chefs specialize on just one type of food, such as sauces or desserts. A menu is often a recipe of smaller recipes. The same thing occurs in the software industry. A large computer program is written by several people, and each team member is responsible for writing a portion.

Functions provide us a way to *organize a solution into logical pieces* that communicate with each other. A function is basically one self-contained part of a computer program. Over the years, computer scientists have used various words to describe this concept: besides the word “function” you may also hear of a “procedure,” “sub-program,” or “method.” They all essentially mean the same thing, and the nuances are not important at this point.

Another purpose of a function in a computer program is to encapsulate some code that might need to be used several times. We *avoid having to retype the same code* in more than one place in the program.

Here is an analogy to illustrate: When you hear a song, there is usually a portion that gets repeated several times, and this turns out to be the most memorable part of the song. It’s called the chorus. And when you see the lyrics of the song printed out or as a computer file, you can see the words for the chorus printed near the top of the lyrics. However, between each of the verses of the song, there may be a special notation [Chorus] in brackets. This means “the chorus goes here.” Why did the person who typed the lyrics decide to write [Chorus] instead of typing out the lyrics of the chorus again?

The reason is to save printed space. It is unnecessary to retype the chorus because it doesn’t change during the song. When you are first learning the song, and you encounter [Chorus] in the lyrics, all you have to do is glance back at the top of the sheet to find the words to the Chorus. And then what do you do when you are finished with the chorus a second time? You need to remember where you left off in the rest of the song.

Exactly the same thing is going on in computer programs that contain functions.

Basic concepts regarding functions

Actually, you have already been using Pascal's built-in functions. Now we get to write our own. The first thing to understand about functions is where they fit into your program.

In Pascal, we *define* functions near the top of a program. After defining all the functions we wish to write, this is followed by the "main program." When you run a program, it begins with the first statement of the main program. We only enter a function if and when it is called. Most of the time, a function is called from somewhere in the main program. But it is possible for one function to call another.

When you write a function, it is important to preface it with a comment, explaining its purpose, and how it works, just like you would include a general comment about an entire program.

A function begins with its declaration. The general format of a function declaration is

`function` *FunctionName* (*formal parameter list*) : *return type*;

For example:

```
function SumSquares (a, b : integer) : integer;
```

The formal parameter list inside the parentheses is similar to a set of variable declarations. They indicate what value(s), if any, are being passed into the function. Finally, notice that the function declaration ends with a semicolon.

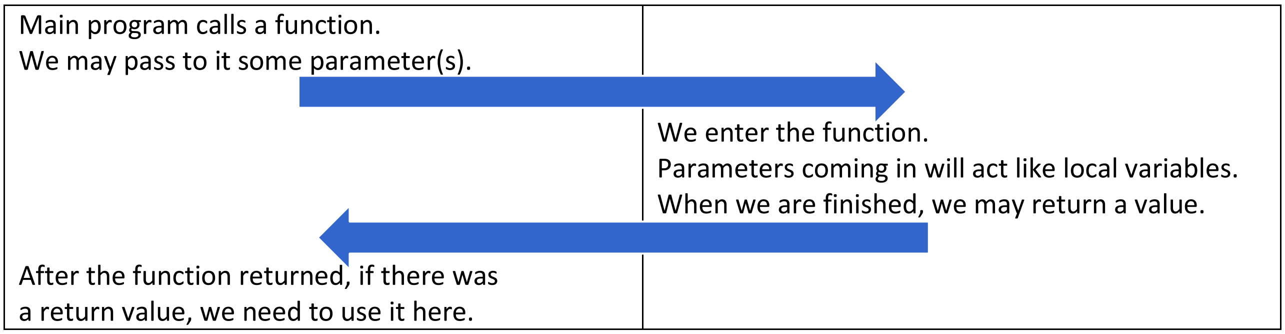
Functions often communicate data with the main program. Usually we pass *parameters* to a function. Parameters are like a form of "input" to a function. Inside the function, we are free to create any "local" variables to assist us in our calculations. Finally, a function usually needs to *return* some data back to the place where the function was called. A function can have any number of parameters, but it can only return one value.

There is one subtlety to note about parameters inside a function. Any changes we make to them inside the function will have no effect once the function is finished. Consider this example. The function and main program are written here side by side for clarity, though in reality the function would be on top.

```
(* main program *)
program example;
var a : integer;
begin
  a := 4;
  fun(a);
  writeln(a);
end.
```

```
(* the function *)
function fun(n : integer): integer;
begin
  n := n + 1;
  writeln(n);
  fun := n;
end;
```

In this example, since a = 4, the value 4 is passed to the function into the parameter n. So, inside the function, the value of n is initially 4. Then we increment it to 5 and print this out. When the function returns to the main program, a is still 4. So, when we print a, we print the number 4. Incrementing n inside the function had no effect on a in the main program.




Let's look at some examples. In each case, the function being defined appears in boldface.

Example #1

```
(* A program to compute a coach's winning percentage.
 * The function will calculate a percentage, assuming
 * that a tie is half a win.
 *)
program example1;

var
  percentage : real;

function calc_record(win, loss, tie : integer) : real;
  begin
    calc_record := (win + 0.5 * tie) / (win + loss + tie);
  end;

(* Main program *)
begin
  percentage := calc_record(10, 5, 3);
  writeln('Coach had a record of ', percentage : 5:3);
end.
```

Example #2

```
(* Let's find the sum of some ranges of integers.
 * Function find_sum will find the sum from low..high
 *)
```

```
program example2;

function find_sum(low, high : integer): integer;
  var
    i, sum : integer;
  begin
    sum := 0;
    for i := low to high do
      sum := sum + i;
    find_sum := sum;
  end;

(* Main program *)
begin
  writeln(find_sum(10, 25));
  writeln(find_sum(4, 72));
  writeln(find_sum(75, 150));
end.
```

Example #3

```
(* Let's calculate the absolute value of the difference
 * as a function.
 *)
program example3;

function abs_val_diff(a, b : integer) : integer;
  begin
    if a > b then
      abs_val_diff := a - b
    else
      abs_val_diff := b - a;
  end;

(* Main program *)
begin
  writeln(abs_val_diff(10, -5));
  writeln(abs_val_diff(14, 18));
end.
```

Example #4

```
(* A program to input a user's first and last name. *)
program example4;
var
  user : string;
```

```
function get_name() : string;
  var
    first, last : string;
  begin
    write('Please enter your first name:  ');
    readln(first);
    write('Please enter your last name:  ');
    readln(last);
    get_name := first + ' ' + last;
  end;

(* Main program *)
begin
  user := get_name();
  writeln('Hello, ', user);
end.
```

Example #5: We convert a Celsius temperature to Fahrenheit using a function.

```
program example5;

(* Conversion function *)
function convert_to_f(celsius_temp : real): real;
  begin
    convert_to_f := 9/5 * celsius_temp + 32;
  end;

(* --------------------------------
 * Main program
 *)
var
  c, f: real;
begin
  write('Enter a Celsius temperature:  ');
  readln(c);
  f := convert_to_f(c);

  writeln('The Fahrenheit temperature is ', f : 5:1);
end.
```

Discussion: For each of the above example programs, answer these questions.

1. In the program, what is the name of the function?
2. How many times is the function called?
3. Does the function take any parameters? If so, how many, and what is/are the type(s) of the parameters?

4. Does the function have any local variables (i.e. variables declared inside the function)?
5. Does the function return a value? If so, what kind of value?

Occasionally when designing a function, we realize that we are not trying to compute some final result. There is no value that needs to be returned. In this case, we define a procedure instead of a function. A procedure is basically the same as a function except it does not return any value. Notice that the format of a procedure definition is similar to that of a function:

procedure `ProcedureName` (*formal parameter list*);

As in this example:

```
procedure PrintResult (x, y : real);
```

Here are two programs that make use of a procedure.

```
(* A program that can count multiple times.
 *
 * count_up - This procedure will print the numbers
 * from 1 up to some given maximum.
 *)
program example6;

procedure count_up(maximum : integer);
  var
    i : integer;
  begin
    for i := 1 to maximum do
      writeln(i);
  end;

(* Main program:  let's count to 10, then to 20. *)
begin
  count_up(10);
  count_up(20);
end.
```

```
(* hubbard.pas - Let's practice writing lines reminiscent of
 * "Old Mother Hubbard" to illustrate procedure calling.
 *)
program Hubbard;
```

```
procedure substitute(place, thing, action : string);
  begin
    writeln('She went to the ', place);
    writeln('To buy him ', thing);
    writeln('But when she came back, ');
    writeln('He ', action, '.');
    writeln();
  end;

begin
  writeln('Old Mother Hubbard');
  writeln('Went to the cupboard,');
  writeln('To give the poor dog a bone;');
  writeln('But when she came there');
  writeln('The cupboard was bare,');
  writeln('And so the poor dog had none.');
  writeln();

  substitute('alehouse', 'some beer', 'sat in a chair');
  substitute('tailor''s', 'a coat', 'was riding a goat');
  substitute('hatter''s', 'a hat', 'was feeding her cat');
  substitute('barber''s', 'a wig', 'was dancing a jig');
  substitute('cobbler''s', 'some shoes', 'was reading the news');
end.
```

It is customary to place all of your defined procedures and functions after your variable declarations but before the main program. Thus, a program that features functions and procedures will have this overall structure.

1. Introductory comment
2. Global constant declarations (const …)
3. Global variable declarations (var …)
4. Procedure and function declarations
   (Each procedure and function may additionally declare its own local variables.)
5. Main program (begin … end.)

Please note that there are two parts to writing any function:

First, we need to *define* the function itself. As we design the function, we need to consider the following.

- Does the function need parameters? If so, how many, and what should they be called?
- What calculation does the function need to do? Is any I/O necessary?
- What should the function return? If you realize that your function does not need to return a value, then you need to declare it as a procedure, rather than as a function.

The second step is to actually *use* the function elsewhere in the program. In other words, somewhere in the main program we should call the function. Here are some considerations.

- What values should we pass to the function?
- What should we do with the answer that the function returns to us? For example, it turns out that a function call is often on the right side of an assignment statement.

Parameters and return value

Functions return a value. Procedures do not. In addition, functions and procedures may or may not need parameters. Therefore, there are 4 possible scenarios that we should consider. They are all plausible, but each one describes a completely different purpose of the function or procedure:

| | |
|---|---|
| 1. Function with parameter(s)<br>Send data, receive data<br><br>This is the most common scenario.<br><br>The function is calculating something for the main program. | 2. Procedure with parameter(s)<br>Send data, but receive no data<br><br>Most likely because we want to output something or write to a file while inside the function. The procedure is not trying to perform a calculation that we need later. |
| 3. Function with no parameters<br>Send no data, but receive data<br><br>Most likely because the function is getting data from the user, an input file, or is generating random data – and this is needed by the rest of the program. | 4. Procedure with no parameters<br>Send no data, receive no data<br><br>No data to be sent in either direction.<br>Usually this means we are just printing a message, or we are doing some isolated step of an overall algorithm. |

What do function and procedure calls look like? Since procedures never return a value, a procedure call is a complete Pascal statement by itself. But functions do return a value, and you need to do something with it. “Do something” means that the function call needs to be on the right side of an assignment, or it needs to be nested inside another function or procedure call as a parameter.

Here are some examples of what function and procedure calls look like:

| Function call | Meaning |
|---|---|
| `proc(25);` | Go to the procedure called proc() and take the number 25 with you. There is no return value to bring back. |
| `proc(fun(25));` | The value returned by fun(25) is passed to a procedure. |
| `x := fun(25);` | The value returned by fun(25) is put into the variable x. |
| `writeln(fun(25));` | The value returned by fun(25) is printed. |

The first two examples above are procedure calls. We pass a value to a procedure. That value is used by the procedure, but the procedure does not return any value back to us. If it did, it would be thrown away! The second example is interesting because we call a function, and its return value is passed to the procedure. The last two examples show typical things we do with a function's return value: assign it to a variable or print it out.

Looking back at how we have used the built-in routines `read`, `readln`, `write` and `writeln`, they have always been used as stand-alone statements. That is because they are all procedures, not functions.

It is a common mistake to write a function call like "fun(25)" when the function in fact returns a value. The following example program illustrates the mistake of <u>discarding a return value</u>. The program contains a function that performs a simple calculation.

```
(* discard.pas - What happens when we accidentally throw away the
 * value returned by a function?
 *)
program discard;

var
  value, result : integer;

function f (x : integer) : integer;
  begin
    f := 3 * x * x - 7;
  end;

(* main program *)
begin
  write('Please enter an integer:  ');
  readln(value);
  f(value);
  writeln('After applying the function, the answer is ', result);
end.
```

This program has a logical error. It will not print the correct result. Run it and see! It's wrong because the variable `result` was never assigned. The function call looks like a procedure call. The way to fix the error is to change the statement

```
  f(value);
```

to this:

```
  result := f(value);
```

Recap

Let's review some important facts about functions and procedures:

- We have already been using built-in procedures and functions like `writeln` and `StrToInt`. What is new in this section is that you are creating your own.
- Procedures and functions you define should be placed immediately before the main program.
- A procedure is a function that does not return a value.
- Functions and procedures may have any number of parameters, including none. It just depends on the purpose of the function or procedure.
- If you call a function or procedure without arguments, then the parentheses are technically optional. But I recommend you include them anyway to improve the readability of your program.
- Variables declared inside a function (or procedure) are "local" only, and cannot be used outside the function. Similarly, changes to variables inside a function have no effect outside the function.
- You can only return one value from a function, such as a single integer, real number, boolean or string.
- You return a value from a function by writing an assignment statement, and putting the function's name on the left side of the :=.

Any of the practice problems we saw earlier can be written as a function instead of a program. Try it! Let me suggest:

- Put the algorithm's calculations inside the function. The function should not do I/O itself. The function's "input" is really the parameters, and its "output" is the result it returns. If you'd like, you can write a separate function to obtain the interactive input from the user and/or a separate function to display the output.
- The main program should pass the appropriate (input) data to the function. And most importantly, it needs to use the function's return value to output to the user.

Exercise:

1. Consider the following Pascal program. Trace its execution by hand and explain what happens. What is the output?

```
program mystery;

function fun(x : integer): integer;
  begin
    fun := 2*x - 1;
  end;
```

```
function fun2(x, y : integer): integer;
  begin
    fun2 := fun(x) + y;
  end;

function getNumber(): integer;
  var
    value : integer;
  begin
    write('Please enter an integer:  ');
    readln(value);
    getNumber := value;
  end;

(* main program *)
var
  a, b, c, d : integer;
begin
  a := 3;
  b := fun(a);
  writeln(b);

  c := fun2(a, b);
  writeln(c);

  d := getNumber();
  writeln(d);
end.
```

2. Consider the following Pascal program. Explain what it accomplishes.

```
program mystery2;

function compute(n : integer): integer;
  begin
    compute := n + 2;
  end;

(* main program *)
var
  i : integer;
  A : array [1..5] of integer = ( 12, 87, 5, 13, 94 );
begin
  for i := 1 to 5 do
    writeln('The result for ', A[i], ' is ', compute(A[i]));
end.
```

3. Let's create a short program that includes the use of a function. The program will ask the user to enter the freezing and boiling points of a substance. We need to convert these values from Fahrenheit to Celsius.

   First, we need a conversion function. The parameter is the Fahrenheit temperature. The return value is the corresponding Celsius temperature.

   The main program will do the following steps: It should ask the user for the freezing and boiling temperatures in Fahrenheit. Next, it needs to call the convert function on the freezing temperature to convert it to Celsius. Then it should do the same for the boiling temperature. Finally, output both Celsius temperatures.

4. Design a program that converts an amount of money from US dollars into euros. Use a function to perform the actual conversion, and do all the I/O in the main program. Assume that 1 Euro is worth $1.13.

5. Design a function that calculates a discounted price. It will need two parameters: a regular price and a discount percentage. The function will return the discounted price. Call the function discount. For example, when we say discount(4.50, 20), the return value should be 3.60.

6. Design a function that takes 3 integer parameters and returns the largest number.

7. Design a procedure that takes a string as a parameter. The procedure should print the first and last character of this string. You may assume that the string has at least 2 characters.

8. Design a procedure that takes a string as a parameter. The procedure should print each character of this string, one per line. In other words, the string is being printed vertically.

9. Design a procedure that takes an integer parameter. The procedure should print this number of asterisks, and then finally print a newline.

10. Design a function that takes a string parameter, and returns a string. The purpose of this function is to rotate the characters by one position. The string being returned will be the same as the string that came in, except that the first character will be moved to the end. For example, the word "pickle" will be returned as "icklep".

11. Print out all the prime numbers between 1 and 1000.

12. Write a function called HasPeriod that takes a string parameter. The function will return true if the string contains at least one '.' character. It will return false if no period exists in the string.

13. Write a function round5 that takes an integer parameter, and returns this integer rounded to the nearest 5.

14. Bob is taking a biology class this term. The class is graded on a pass/fail basis. A student needs to earn a grade of at least 60.0 percent to pass. The course grade is the average of the three test scores. Bob wants to know what minimum score he needs on the third test in order to pass the class. Write a function to help him. The function should take two parameters – the scores on the first two tests. The function should return the minimum score needed to achieve a 60.0 average on all three tests. Assume that all test scores are integers. For example, if a student makes 59 and 60 on the first two exams, then the function should return 61.

15. Go back to the algorithm practice problems you saw earlier. Select one of these problems, and write a program to solve it, using a function to perform the essential computation.

16. We can find hierarchical information and multiple "function instances" in everyday life. For example, the nutrition label for Publix's Cool Mint Cookie frozen yogurt shows several levels of nesting. Literally, the ingredients read as follows.

    Cultured lowfat milk, sugar, corn syrup, nonfat dry milk, whey protein concentrate, cultured dairy solids, chocolate fudge [sugar, peanut oil, cocoa (processed with alkai), whey, salt, soy lecithin], choco-coated mint cookies ** {cookie [wheat flour, sugar, partially hydrogenated soybean oil and/or cottonseed oil, cocoa processed with alkai, soy lecithin (an emulsifier), salt, sodium bicarbonate], sugar, coconut oil, cocoa processed with alkai, cocoa, nonfat milk, palm kernel oil, milkfat, natural flavors, soy lecithin, oil of peppermint, color added (yellow 5 lake, blue 1 lake)}, vegetable mono and diglycerides, natural flavors, locust bean gum, guar gum, sodium citrate, calcium sulfate, carrageenan and peppermint extract.

    ** manufactured in a facility that produces peanuts, tree nuts and milk products.

    Copy and paste the above ingredient list to a new blank text file. Separate the list out so that only one ingredient or subingredient appears per line. Indent the ingredients to show the levels of nesting. The above ingredient list uses various types of grouping symbols such as parentheses, brackets and braces. Place each grouping symbol by itself on a line to make the hierarchy easier to read.

    Which foodstuffs are mentioned more than once in the list of ingredients? This would be analogous to a function that is called from more than one place in a computer program.

Congratulations! This is the end of the first unit of the course. As we turn our attention to specific applications of problem solving, you will have more opportunities to practice your craft in the lab and learn a little more Pascal as needed along the way.

Links to some Pascal documentation

- Features of the language: https://wiki.freepascal.org/Contents
- Built-in routines: https://www.freepascal.org/docs-html/rtl/system/index-5.html

For example, some of the most interesting features of Pascal we did not explore in this document include: repeat-until loops, var parameters, multi-dimensional arrays, random numbers and records.