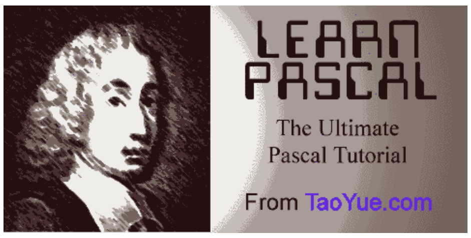
LEARN PASCAL

Index

Introduction



Welcome to *Learn Pascal*! This tutorial is an introduction to the Pascal simple, yet complete, introduction to the Pascal programming language. It covers all of the syntax of standard Pascal, including pointers.

I have tried to make things are clear as possible. If you don't understand anything, try it in your Pascal compiler and tweak things a bit. Pascal was designed for teaching purposes, and is a very structured and syntactically-strict language. This means the compiler will catch more beginner errors and yield more beginner-friendly error messages than with a shorthand-laden language such as C or PERL.

This tutorial was written for *beginner* programmers, so assumes no knowledge. At the same time, a surprising number of experienced programmers have found the tutorial a useful reference source for picking up Pascal.

We begin with some background on Pascal, an explanation of compilers, and step-by-step instructions for getting one such compiler working on a modern Windows operating system. The background section is informative reading, I'm told, for experienced programmers as well as novices, but the Table of Contents will let you pick any topic if you're already familiar with programming.

History of Pascal

Origins

Pascal grew out of ALGOL, a programming language intended for scientific computing. Meeting in Zurich, an international committee designed ALGOL as a platform-independent language. This gave them comparatively free rein in the features they could design into ALGOL, but also made it more difficult to write compilers for it. Those were the days when many computers lacked hardware features that we now take for granted. The lack of compilers on many platforms, combined with its lack of pointers and many basic data types such as characters, led to ALGOL not being widely accepted. Scientists and engineers flocked to FORTRAN, a programming language which *was* available on many platforms. ALGOL mostly faded away except as a language for describing algorithms.

Wirth Invents Pascal

In the 1960s, several computer scientists worked on extending ALGOL. One of these was Dr. Niklaus Wirth of the Swiss Federal Institute of Technology (ETH-Zurich), a member of the original group that created ALGOL. In 1971, he published his specification for a highly-structured language which resembled ALGOL in many ways. He named it *Pascal* after the 17th-century French philosopher and mathematician who built a working mechanical digital computer.
Pascal is very data-oriented, giving the programmer the ability to define custom data types. With this freedom comes strict type-checking, which prevented data types from being mixed up. Pascal was intended as a teaching language, and was widely adopted as such. Pascal is free-flowing, unlike FORTRAN, and reads very much like a natural language, making it very easy to understand code written in it.

UCSD Pascal

One of the things that killed ALGOL was the difficulty of creating a compiler for it. Dr. Wirth avoided this by having his Pascal compiler compile to an intermediate, platform-independent object code stage. Another program turned this intermediate code into executable code.
Prof. Ken Bowles at the University of California at San Diego (UCSD) seized on the opportunity this offered to adapt the Pascal compiler to the Apple II, the most popular microcomputer of the day. UCSD P-System became a standard, and was widely used at universities. This was aided by the low cost of Apple II's compared to mainframes, which were necessary at the time to run other languages such as FORTRAN. Its impact on computing can be seen in IBM's advertisements for its revolutionary Personal Computer, which boasted that the PC supported three operating systems: Digital Research's CP/M-86, Softech's UCSD P-system, and MicroSoft's PC-DOS.

Pascal Becomes Standard

By the early 1980's, Pascal had already become widely accepted at universities. Two events conspired to make it even more popular.
First, the Educational Testing Service, the company which writes and administers the principal college entrance exam in the United States, decided to add a Computer Science exam to its Advanced Placement exams for high school students. For this exam, it chose the Pascal language. Because of this, secondary-school students as well as college

students began to learn Pascal. Pascal remained the official language of the AP exams until 1999, when it was replaced by C++, which was quickly replaced by Java.
Second, a small company named Borland International released the Turbo Pascal compiler for the IBM Personal Computer. The compiler was designed by Anders Hejlsberg, who would later head the group at Microsoft that developed C# and (re)introduced Managed Code back to the world of computing.
Turbo Pascal was truly revolutionary. It did take some shortcuts and made some modifications to standard Pascal, but these were minor and helped it achieve its greatest advantage: speed. Turbo Pascal compiled at a dizzying rate: several thousand lines a minute. At the time, the available compilers for the PC platform were slow and bloated. When Turbo Pascal came out, it was a breath of fresh air. Soon, Turbo Pascal became the *de facto* standard for programming on the PC. When *PC Magazine* published source code for utility programs, it was usually in either assembly or Turbo Pascal.
At the same time, Apple came out with its Macintosh series of computers. As Pascal was the preeminent structured programming language of the day, Apple chose Pascal as the standard programming language for the Mac. When programmers received the API and example code for Mac programming, it was all in Pascal.

Extensions

From version 1.0 to 7.0 of Turbo Pascal, Borland continued to expand the language. One of the criticisms of the original version of Pascal was its lack of separate compilation for modules. Dr. Wirth even created a new programming language, Modula-2, to address that problem. Borland added modules to Pascal with its units feature.
By version 7.0, many advanced features had been added. One of these was DPMI (DOS Protected Mode Interface), a way to run DOS programs in protected mode, gaining extra speed and breaking free of the 640K barrier for accessing memory under DOS. Turbo Vision, a text-based windowing system, allowed programmers to create great-looking interfaces in practically no time at all. Pascal even became object-oriented, as version 5.5 adopted the Apple Object Pascal extensions. When Windows 3.0 came out, Borland created Turbo Pascal for Windows, bringing the speed and ease of Pascal to the graphical user interface. It seemed that Pascal's future was secure.

The World Changes

However, this was not to be. In the 1970s, Dennis Ritchie and Brian Kernighan of AT&T Bell Laboratories created the C Programming Language. Ritchie then collaborated with Ken Thompson to design the UNIX operating system. At the time, AT&T had a government-sanctioned monopoly on telephone service in the United States. In return for the monopoly, its telephone business was regulated and it was prohibited from entering the computer business. AT&T, seeing no market for a research operating system, gave UNIX away to universities for free, complete with source code.
Thus, a whole generation of computer science students learned C in their university courses on languages and operating systems. Slowly but surely, C began to filter into the computer programming world.
Pascal was finally killed by object orientation and the move to Windows on the industry-standard PC platform. In the 1980s, Bjarne Stroustrop, also of Bell Labs, popularized object-orientation by developing C++, which kept the familiar syntax of C while extending it for object orientation. C++ came to define object orientation to a generation of programmers, and remains a strong force even today.
Also in the 1980s, Microsoft Windows adopted C as its standard programming language. In contrast to MacOS and Pascal, the Windows API samples were all in K&R (pre-ANSI)

C, complete with variable lists after the function prototype. As object orientation and Windows took hold, the natural language for applications migrating to Windows was C++. Many colleges and universities moved away from Pascal, choosing C++ or newer languages for their programming courses. Finally, the AP exam moved to C++, ending Pascal's dominance in American high schools.

So Why Learn Pascal?

Despite its fading away as a *de facto* standard, Pascal is still quite useful. C and C++ are very symbolic languages. Where Pascal chooses words (e.g. **begin**-**end**), C/C++ instead uses symbols (**{**-**}**). Also, C was designed for systems programming. In Pascal, mixing types leads to an error and is very infrequently done. In C/C++, type-casting and pointer arithmetic is common, making it easy to crash programs and write in buffer overruns. When the AP exam switched to C++, only a subset of C++ was adopted. Many features, like arrays, were considered too dangerous for students, and ETS provided its own "safe" version of these features.

Another reason: speed and size. The Borland Pascal compiler is still lightning-fast. Borland has revitalized Pascal for Windows with Delphi, a Rapid-Application-Development environment. Instead of spending several hours writing a user interface for a Windows program in C/C++, you could do it in ten minutes with Delphi's graphical design tools. Delphi is to Pascal what Visual BASIC did to BASIC. Borland is still developing Delphi, and the open-source community has created a largely Borland-compatible compiler called Free Pascal.

Also, Pascal remains preferred at many universities, especially in areas where students are first exposed to computers at school rather than at home. In addition, Pascal was well-suited for teaching programming, and remains so. There is less overhead and fewer ways for a student to get a program into trouble. For teaching simple procedural programming, Pascal remains a good choice. Pascal has hung on longer in education outside the United States, and remains an official language of the International Informatics Olympiad. A basic programming background is useful in many technical occupations, and the overhead of learning an object-oriented language is not necessarily the best application of resources. Thus, even after C, C++, and Java took over the programming world, Pascal retains a niche in the market. Many small-scale freeware, shareware, and open-source programs are written in Pascal/Delphi. So enjoy learning it while it lasts. It's a great introduction to computer programming. It's not scary like C, dangerous like C++, or abstract like Java. In another twenty years, you'll be one of the few computer programmers to know and appreciate Pascal.

Pascal Compilers

This document will explain the basics about compilers as well as provide links to well-known Pascal compilers and explain how to set up Free Pascal.

About Computer Languages and Compilers

When talking about computer languages, there are basically three major terms that will be used.

1. **Machine language** -- actual binary code that gives basic instructions to the computer's CPU. These are usually very simple commands like adding two numbers or moving data from one memory location to another.
2. **Assembly language** -- a way for humans to program computers directly without memorizing strings of binary numbers. There is a one-to-one correspondance with machine code. For example, in Intel x86 machine language, `ADD` and `MOV` are mnemonics for the addition and move operations.
3. **High-level language** -- permits humans to write complex programs without going step-by step. High-level languages include Pascal, C, C++, FORTRAN, Java, BASIC, and many more. One command in a high-level language, like writing a string to a file, may translate to dozens or even hundreds of machine language instructions.

Microprocessors can only run machine language programs directly. Assembly language programs are *assembled*, or translated into machine language. Likewise, programs written in high-level languages, like Pascal, must also be translated into machine language before they can be run. To do this translation is to *compile* a program.

The program that accomplishes the translation is called a ***compiler***. This program is rather complex since it not only creates machine language instructions from lines of code, but often also optimizes the code to run faster, adds error-correction code, and links the code with subroutines stored elsewhere. For example, when you tell the computer to print something to the screen, the compiler translates this as a call to a pre-written module. Your code must then be linked to the code that the compiler manufacturer provides before an executable program results.

With high-level languages, there are again three basic terms to remember:

1. **Source code** -- the code that *you* write. This typically has an extension that indicates the language used. For example, Pascal source code usually ends in ".pas" and C++ code usually ends in ".cpp"
2. **Object code** -- the result of compiling. Object code usually includes only one module of a program, and cannot be run yet since it is incomplete. On DOS/Windows systems, this usually has an extension of ".obj"
3. **Executable code** -- the end result. All the object code modules necessary for a program to function are linked together. On DOS/Windows systems, this usually has an extension of ".exe"

More About Compilers

The *de facto* standard in DOS and Windows-based compilers is Borland Pascal. Before it came out, most Pascal compilers were clumsy and slow, strayed from the Pascal standard, and cost several hundred dollars. In 1984, Borland introduced Turbo Pascal, which sold for less than $100, compiled an order of magnitude faster than existing compilers, and came with an abundance of source code and utility programs.

This product was a great success and was prominent for almost a decade. But in the 1990s, the world was moving to Windows. In 1993, the last version of Turbo Pascal, version 7 for DOS, came out. After that, the demand for DOS programs plummetted and Borland (renamed Inprise, then back to Borland) focused on producing Windows compilers.

This tutorial will only deal with console-based programming, where the computer prints lines of data to the screen and the user interacts with the program using a keyboard. The goal of the tutorial is to teach *how* to program in Pascal. Once you've learned that, you can easily look at a reference book or another web page and pick up graphics and windowing systems on your own.

Although old commercial Pascal compilers are often available for download, Turbo Pascal 5.5 from the Borland Museum and Symantec Think Pascal (Macintosh) linked from The Free Country's Free Pascal Compiler List, computers have progressed much since the 1980s and early 1990s. We are no longer stuck with 8.3 filenames on DOS or non-preemptive multitasking on Mac OS. Using an old compiler is fun in the same sense as playing an old game on an emulator, but the open source movement has produced good compilers for modern operating systems, and a beginner will find it much easier to use those.

Open Source Compilers

The two main open-source compiler projects are:

- GNU Pascal
- Free Pascal

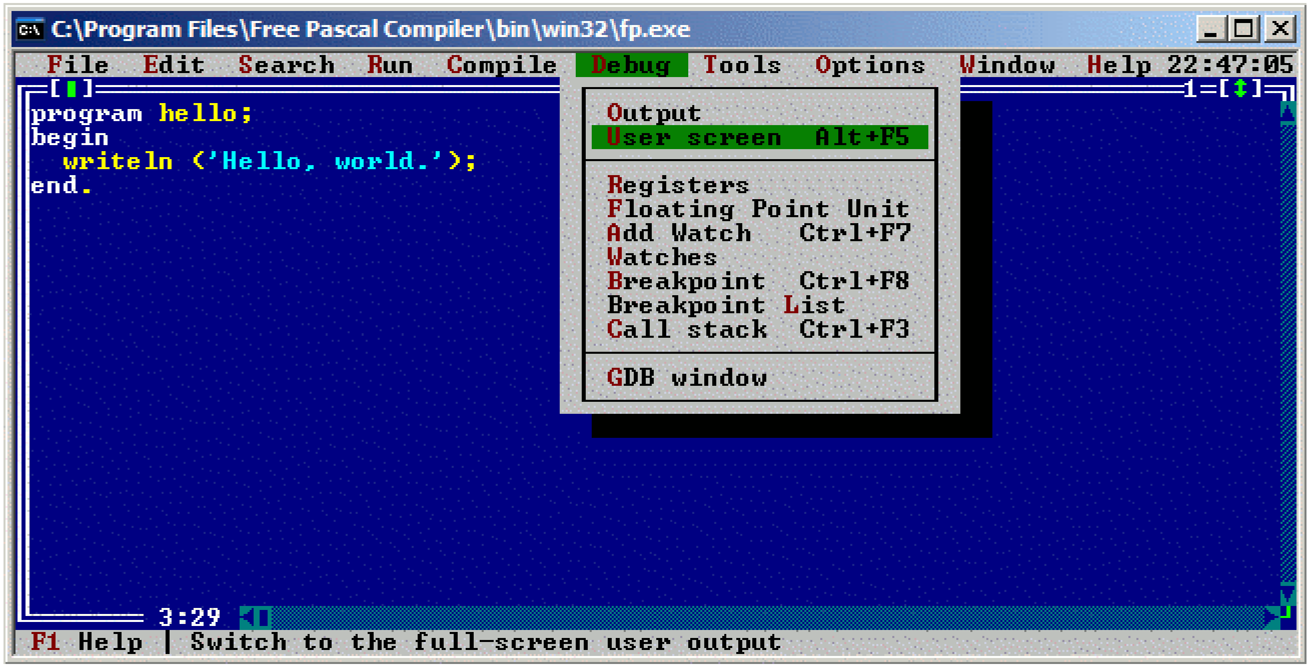
Free Pascal is generally considered friendlier for novices, and strives to emulate Borland Pascal in many ways, though both will serve fine for learning Pascal.

As most users of this tutorial will be running Windows, here's how to set up Free Pascal and get to the point where you're compiling a program on a modern Windows operating system:

1. Download the Win32 installer for Free Pascal from the Free Pascal download page.
2. Run the file you just downloaded and go through the wizard to setup Free Pascal.
3. Open Free Pascal using the shortcut (by default it is located in Start → Free Pascal.
4. Type in a program (flip to the next lesson to get a "Hello, world." program).
5. Save the file with `File-Save As ...`
6. Run the program from the `Run` menu. This will automatically compile the program if you've made any changes, then run the program. It will also run the program without compiling if you've not made any changes since the last time you compiled.

With programs that don't expect user input, you'll see the program flash on a black screen. But the program completes in the blink of an eye and you are returned to the IDE without seeing the results of your work. There are two ways around this:

- Select `User screen` from the `Debug` menu to see the results of the program.
- Add a `readln` statement at the end of every program. This will make the program wait for the user to press the *Enter* key before the program ends and returns to the IDE.


C:\Program Files\Free Pascal Compiler\bin\win32\fp.exe

```
program hello;
begin
  writeln ('Hello, world.');
end.
```

Note that a `.exe` file was created in the directory where you saved your program. This is the executable. You can go to the Command Prompt, change to the directory, and run this executable straight. You can also double-click on it in Windows Explorer (and it will still flash by quickly if it ends without requiring user input).

Hello, world.

In the short history of computer programming, one enduring tradition is that the first program in a new language is a "Hello, world" to the screen. So let's do that. Copy and paste the program below into your IDE or text editor, then compile and run it.

If you have no idea how to do this, return to the Table of Contents. Earlier lessons explain what a compiler is, give links to downloadable compilers, and walk you through the installation of an open-source Pascal compiler on Windows.

```
program Hello;
  begin (* Main *)
  writeln ('Hello, world.')
end. (* Main *)
```

The output on your screen should look like:

```
Hello, world.
```

If you're running the program in an IDE, you may see the program run in a flash, then return to the IDE before you can see what happened. See the bottom of the previous lesson for the reason why. One suggested solution, adding a `readln` to wait for you to press *Enter* before ending the program, would alter the "Hello, world" program to become:

```
program Hello;
begin (* Main *)
  writeln ('Hello, world.');
  readln
end. (* Main *)
```

Basics

The basic structure of a Pascal program is:

```
PROGRAM ProgramName (FileList);

CONST
  (* Constant declarations *)

TYPE
  (* Type declarations *)

VAR
  (* Variable declarations *)

(* Subprogram definitions *)

BEGIN
  (* Executable statements *)
END.
```

The elements of a program must be in the correct order, though some may be omitted if not needed. Here's a program that does nothing, but has all the required elements:

```
program DoNothing;
begin
end.
```

Comments are portions of the code which do not compile or execute. Pascal comments start with a `(*` and end with a `*)`. You cannot nest comments:

```
    (* (* *) *)
```

will yield an error because the compiler matches the first `(*` with the first `*)`, ignoring the second `(*` which is between the first set of comment markers. The second `*)` is left without its matching `(*`. This problem with begin-end comment markers is one reason why many languages use line-based commenting systems.

Turbo Pascal and most other modern compilers support brace comments, such as `{Comment}`. The opening brace signifies the beginning of a block of comments, and the ending brace signifies the end of a block of comments. Brace comments are also used for compiler directives.

Commenting makes your code easier to understand. If you write your code without comments, you may come back to it weeks, months, or years later without a guide to why you coded the program that way. In particular, you may want to document the major design of your program and insert comments in your code when you deviate from that design for a good reason.

In addition, comments are often used to take problematic code out of action without deleting it. Remember the earlier restriction on nesting comments? It just so happens that braces {} take precedence over parentheses-stars (* *). You will not get an error if you do this:

```
{ (* Comment *) }
```

Whitespace (spaces, tabs, and end-of-lines) are ignored by the Pascal compiler unless they are inside a literal string. However, to make your program readable by human beings, you should indent your statements and put separate statements on separate lines. Indentation is often an expression of individuality by programmers, but collaborative projects usually select one common style to allow everyone to work from the same page.

Identifiers are names that allow you to reference stored values, such as variables and constants. Also, every program and unit must be named by an identifier.

Rules for identifiers:

- Must begin with a letter from the English alphabet.
- Can be followed by alphanumeric characters (alphabetic characters and numerals) and possibly the underscore (_).
- May not contain certain special characters, many of which have special meanings in Pascal.
  ~ ! @ # $ % ^ & * ( ) + ` - = { } [ ] : " ; ' < > ? , . / |

Different implementations of Pascal differ in their rules on special characters. Note that the underscore character (_) is usually allowed.

Several identifiers are *reserved* in Pascal as syntactical elements. You are not allowed to use these for your identifiers. These include but are not limited to:

and array begin case const div do downto else end file for forward function goto if in label mod nil not of or packed procedure program record repeat set then to type until var while with

Modern Pascal compilers ship with much functionality in the API (Application Programming Interfaces). For example, there may be one unit for handling graphics (e.g. drawing lines) and another for mathematics. Unlike newer languages such as C# and Java, Pascal does not provide a classification system for identifiers in the form of namespaces. So each unit that you use may define some identifiers (say `DrawLine`) which you can no longer use. Pascal includes a *system unit* which is automatically used by all programs. This provides baseline functionality such as rounding to integer and calculating logarithms. The system unit varies among compilers, so check your documentation. Here is the system unit documentation for Free Pascal Compiler.

Pascal is *not* case sensitive! (It was created in the days when all-uppercase computers were common.) `MyProgram`, `MYPROGRAM`, and `mYpRoGrAm` are equivalent. But for readability purposes, it is a good idea to use meaningful capitalization. Most programmers will be on the safe side by *never* using two capitalizations of the same identifiers for different purposes, regardless of whether or not the language they're using is case-sensitive. This reduces confusion and increases productivity.

Identifiers can be any length, but some Pascal compilers will only look at the first several characters. One usually does not push the rules with extremely long identifiers or loads of special characters, since it makes the program harder to type for the programmer. Also, since most programmers work with many different languages, each with different rules about special characters and case-sensitivity, it is usually best to stick with alphanumeric characters and the underscore character.

Constants are referenced by identifiers, and can be assigned one value at the beginning of the program. The value stored in a constant cannot be changed.

Constants are defined in the constant section of the program:

```
const
  Identifier1 = value;
  Identifier2 = value;
  Identifier3 = value;
```

For example, let's define some constants of various data types: strings, characters, integers, reals, and Booleans. These data types will be further explained in the next section.

```
const
  Name = 'Tao Yue';
  FirstLetter = 'a';
  Year = 1997;
  pi = 3.1415926535897932;
  UsingNCSAMosaic = TRUE;
```

Note that in Pascal, characters are enclosed in single quotes, or apostrophes (`'`)! This contrasts with newer languages which often use or allow double quotes or Heredoc notation. Standard Pascal does not use or allow double quotes to mark characters or strings.

Constants are useful for defining a value which is used throughout your program but may change in the future. Instead of changing every instance of the value, you can change just the constant definition.

Typed constants force a constant to be of a particular data type. For example,

```
const
  a : real = 12;
```

would yield an identifier `a` which contains a real value `12.0` instead of the integer value `12`.

Variables are similar to constants, but their values can be changed as the program runs. Variables must first be declared in Pascal before they can be used:

```
var
  IdentifierList1 : DataType1;
  IdentifierList2 : DataType2;
  IdentifierList3 : DataType3;
  ...
```

`IdentifierList` is a series of identifiers, separated by commas (`,`). All identifiers in the list are declared as being of the same data type.

The basic data types in Pascal include:

- `integer`
- `real`
- `char`

- `Boolean`

Standard Pascal does not make provision for the `string` data type, but most modern compilers do. Experienced Pascal programmers also use pointers for dynamic memory allocation, objects for object-oriented programming, and many others, but this gets you started.

More information on Pascal data types:

- The **`integer`** data type can contain integers from -32768 to 32767. This is the signed range that can be stored in a 16-bit word, and is a legacy of the era when 16-bit CPUs were common. For backward compatibility purposes, a 32-bit signed integer is a `longint` and can hold a much greater range of values.
- The **`real`** data type has a range from 3.4×10^{-38} to 3.4×10^{38}, in addition to the same range on the negative side. Real values are stored inside the computer similarly to scientific notation, with a mantissa and exponent, with some complications. In Pascal, you can express real values in your code in either fixed-point notation or in scientific notation, with the character `E` separating the mantissa from the exponent. Thus,
  `452.13` is the same as `4.5213e2`
- The **`char`** data type holds characters. Be sure to enclose them in single quotes, like so: `'a' 'B' '+'` Standard Pascal uses 8-bit characters, not 16-bits, so Unicode, which is used to represent all the world's language sets in one UNIfied CODE system, is not supported.
- The **`Boolean`** data type can have only two values:
  `TRUE` and **`FALSE`**

An example of declaring several variables is:

```
var
  age, year, grade : integer;
  circumference : real;
  LetterGrade : char;
  DidYouFail : Boolean;
```

Once you have declared a variable, you can store values in it. This is called *assignment*.

To assign a value to a variable, follow this syntax:

```
variable_name := expression;
```

Note that unlike other languages, whose assignment operator is just an equals sign, Pascal uses a colon followed by an equals sign, similarly to how it's done in most computer algebra systems.

The expression can either be a single value:

```
some_real := 385.385837;
```

or it can be an arithmetic sequence:

```
some_real := 37573.5 * 37593 + 385.8 / 367.1;
```

The arithmetic operators in Pascal are:

Operator	Operation	Operands	Result
+	Addition or unary positive	real or integer	real or integer
-	Subtraction or unary negative	real or integer	real or integer
*	Multiplication	real or integer	real or integer
/	Real division	real or integer	real
div	Integer division	integer	integer
mod	Modulus (remainder division)	integer	integer

div and **mod** only work on integers. **/** works on both reals and integers but will always yield a real answer. The other operations work on both reals and integers. When mixing integers and reals, the result will always be a real since data loss would result otherwise. This is why Pascal uses two different operations for division and integer division. 7 / 2 = 3.5 (real), but `7 div 2 = 3` (and `7 mod 2 = 1` since that's the remainder).

Each variable can only be assigned a value that is of the same data type. Thus, you cannot assign a real value to an integer variable. *However*, certain data types will convert to a higher data type. This is most often done when assigning integer values to real variables. Suppose you had this variable declaration section:

```
var
  some_int : integer;
  some_real : real;
```

When the following block of statements executes,

```
some_int := 375;
some_real := some_int;
```

`some_real` will have a value of 375.0.

Changing one data type to another is referred to as *typecasting*. Modern Pascal compilers support explicit typecasting in the manner of C, with a slightly different syntax. However, typecasting is usually used in low-level situations and in connection with object-oriented programming, and a beginning programming student will not need to use it. Here is information on typecasting from the GNU Pascal manual.

In Pascal, the minus sign can be used to make a value negative. The plus sign can also be used to make a value positive, but is typically left out since values default to positive.

Do not attempt to use two operators side by side, like in:

```
some_real := 37.5 * -2;
```

This may make perfect sense to you, since you're trying to multiply by negative-2. However, Pascal will be confused — it won't know whether to multiply or subtract. You can avoid this by using parentheses to clarify:

```
some_real := 37.5 * (-2);
```

The computer follows an order of operations similar to the one that you follow when you do arithmetic. Multiplication and division (`* / div mod`) come before addition and subtraction (`+ -`), and parentheses always take precedence. So, for example, the value of: `3.5*(2+3)` will be `17.5`.

Pascal cannot perform standard arithmetic operations on Booleans. There is a special set of Boolean operations. Also, you should not perform arithmetic operations on characters.

Pascal has several standard mathematical functions that you can utilize. For example, to find the value of `sin` of *pi* radians:

```
value := sin (3.1415926535897932);
```

Note that the `sin` function operates on angular measure stated in radians, as do all the trigonometric functions. If everything goes well, `value` should become 0.

Functions are called by using the function name followed by the argument(s) in parentheses. Standard Pascal functions include:

Function	**Description**	**Argument type**	**Return type**
abs	absolute value	real or integer	same as argument
arctan	arctan in radians	real or integer	real
cos	cosine of a radian measure	real or integer	real
exp	`e` to the given power	real or integer	real
ln	natural logarithm	real or integer	real
round	round to nearest integer	real	integer
sin	sin of a radian measure	real or integer	real
sqr	square (power 2)	real or integer	same as argument
sqrt	square root (power 1/2)	real or integer	real
trunc	truncate (round down)	real or integer	integer

For ordinal data types (integer or char), where the allowable values have a distinct predecessor and successor, you can use these functions:

Function	**Description**	**Argument type**	**Return type**
chr	character with given ASCII	integer	char

	value		
ord	ordinal value	integer or char	integer
pred	predecessor	integer or char	same as argument type
succ	successor	integer or char	same as argument type

Real is not an ordinal data type! That's because it has no distinct successor or predecessor. What is the successor of `56.0`? Is it 56.1, 56.01, 56.001, 56.0001?

However, for an integer `56`, there is a distinct predecessor — `55` — and a distinct successor — `57`.

The same is true of characters:

```
'b'
Successor: 'c'
Predecessor: 'a'
```

The above is not an exhaustive list, as modern Pascal compilers include thousands of functions for all sorts of purposes. Check your compiler documentation for more.

Since Pascal ignores end-of-lines and spaces, punctuation is needed to tell the compiler when a statement ends.

You *must* have a semicolon following:

- the program heading
- each constant definition
- each variable declaration
- each type definition (to be discussed later)
- almost all statements

The last statement in a `BEGIN-END` block, the one immediately preceding the `END`, does not require a semicolon. However, it's harmless to add one, and it saves you from having to add a semicolon if suddenly you had to move the statement higher up.

Indenting is not required. However, it is of great use for the programmer, since it helps to make the program clearer. If you wanted to, you could have a program look like this:

```
program Stupid; const a=5; b=385.3; var alpha,beta:real; begin alpha := a + b;
beta:= b / a end.
```

But it's much better for it to look like this:

```
program NotAsStupid;

const
```

```
  a = 5;
  b = 385.3;

var
  alpha,
  beta : real;

begin (* main *)
  alpha := a + b;
  beta := b / a
end. (* main *)
```

In general, indent each block. Skip a line between blocks (such as between the const and var blocks). Modern programming environments (IDE, or Integrated Development Environment) understand Pascal syntax and will oten indent for you as you type. You can customize the indentation to your liking (display a tab as three spaces or four?).

Proper indentation makes it much easier to determine how code works, but is vastly aided by judicious commenting.

Now you know how to use variables and change their value. Ready for your first programming assignment?

But there's one small problem: you haven't yet learned how to display data to the screen! How are you going to know whether or not the program works if all that information is still stored in memory and not displayed on the screen?

So, to get you started, here's a snippet from the next few lessons. To display data, use:

```
writeln (argument_list);
```

The argument list is composed of either strings or variable names separated by commas. An example is:

```
writeln ('Sum = ', sum);
```

Here's the programming assignment for Chapter 1:

Find the sum and average of five integers. The sum should be an integer, and the average should be real. The five numbers are: 45, 7, 68, 2, and 34.

Use a constant to signify the number of integers handled by the program, i.e. define a constant as having the value `5`.

Then print it all out! The output should look something like this:

```
Number of integers = 5
Number1 = 45
Number2 = 7
Number3 = 68
Number4 = 2
Number5 = 34
Sum = 156
Average = 3.1200000000E+01
```

As you can see, the default output method for real numbers is scientific notation. Chapter 2 will explain you how to format it to fixed-point decimal.

To see one possible solution of the assignment, go to the next page.

Here's one way to solve the programming assignment in the previous section.

```
(* Author:    Tao Yue
   Date:      19 June 1997
   Description:
      Find the sum and average of five predefined numbers
   Version:
      1.0 - original version
*)

program SumAverage;

const
   NumberOfIntegers = 5;

var
   A, B, C, D, E : integer;
   Sum : integer;
   Average : real;

begin    (* Main *)
   A := 45;
   B := 7;
   C := 68;
   D := 2;
   E := 34;
   Sum := A + B + C + D + E;
   Average := Sum / NumberOfIntegers;
   writeln ('Number of integers = ', NumberOfIntegers);
   writeln ('Number1 = ', A);
   writeln ('Number2 = ', B);
   writeln ('Number3 = ', C);
   writeln ('Number4 = ', D);
   writeln ('Number5 = ', E);
   writeln ('Sum = ', Sum);
   writeln ('Average = ', Average)
end.     (* Main *)
```

Input/Output

Input is what comes into the program. It can be from the keyboard, the mouse, a file on disk, a scanner, a joystick, etc.

We will not get into mouse input in detail, because that syntax differs from machine to machine. In addition, today's event-driven windowing operating systems usually handle mouse input for you.

The basic format for reading in data is:

```
read (Variable_List);
```

Variable_List is a series of variable identifiers separated by commas.

`read` treats input as a stream of characters, with lines separated by a special end-of-line character. `readln`, on the other hand, will skip to the next line after reading a value, by automatically moving past the next end-of-line character:

```
readln (Variable_List);
```

Suppose you had this input from the user, and `a, b, c,` and `d` were all integers.

```
45 97 3
1 2 3
```

Here are some sample `read` and `readln` statements, along with the values read into the appropriate variables.

Statement(s)	a	b	c	d
`read (a);` `read (b);`	45	97		
`readln (a);` `read (b);`	45	1		
`read (a, b, c, d);`	45	97	3	1
`readln (a, b);` `readln (c, d);`	45	97	1	2

When reading in integers, all spaces are skipped until a numeral is found. Then all subsequent numberals are read, until a non-numeric character is reached (including, but not limited to, a space).

```
8352.38
```

When an integer is read from the above input, its value becomes `8352`. If, immediately afterwards, you read in a character, the value would be `'.'` since the read head stopped at the first alphanumeric character.

Suppose you tried to read in two integers. That would not work, because when the computer looks for data to fill the second variable, it sees the `'.'` and stops since it couldn't find any data to read.

With real values, the computer also skips spaces and then reads as much as can be read. However, many Pascal compilers place one additional restriction: a real that has no whole part must begin with `0`. So `.678` is invalid, and the computer can't read in a real, but `0.678` is fine.

Make sure that all identifiers in the argument list refer to variables! Constants cannot be assigned a value, and neither can literal values.

For writing data to the screen, there are also two statements, one of which you've seen already in last chapter's programming assignment:

```
write (Argument_List);
writeln (Argument_List);
```

The `writeln` statement skips to the next line when done.

You can use strings in the argument list, either constants or literal values. If you want to display an apostrophe within a string, use two consecutive apostrophes. Displaying two consecutive apostrophes would then requires you to use four. This use of a special sequence to refer to a special character is called *escaping*, and allows you to refer to any character even if there is no key for it on the keyboard.

Formatting output is quite easy. For each identifier or literal value on the argument list, use:

```
Value : field_width
```

The output is right-justified in a field of the specified integer width. If the width is not long enough for the data, the width specification will be ignored and the data will be displayed in its entirety (except for real values — see below).

Suppose we had:

```
write ('Hi':10, 5:4, 5673:2);
```

The output would be (that's eight spaces before the `Hi` and three spaces after):

```
Hi 55673
```

For real values, you can use the aforementioned syntax to display scientific notation in a specified field width, or you can convert to fixed decimal-point notation with:

```
Value : field_width : decimal_field_width
```

The field width is the *total* field width, including the decimal part. The whole number part is always displayed fully, so if you have not allocated enough space, it will be displayed anyway. However, if the number of decimal digits exceeds the specified decimal field width, the output will be displayed rounded to the specified number of places (though the variable itself is not changed).

```
write (573549.56792:20:2);
```

would look like (with 11 spaces in front):

```
           573549.57
```

Reading from a file instead of the console (keyboard) can be done by:

```
read (file_variable, argument_list);
write (file_variable, argument_list);
```

Similarly with `readln` and `writeln`. *file_variable* is declared as follows:

```
var
  ...
  filein, fileout : text;
```

The `text` data type indicates that the file is just plain text.

After declaring a variable for the file, and before reading from or writing to it, we need to associate the variable with the filename on the disk and open the file. This can be done in one of two ways. Typically:

```
reset (file_variable, 'filename.extension');
     rewrite (file_variable, 'filename.extension');
```

`reset` opens a file for reading, and `rewrite` opens a file for writing. A file opened with *reset* can only be used with *read* and *readln*. A file opened with *rewrite* can only be used with *write* and *writeln*.

Turbo Pascal introduced the *assign* notation. First you assign a filename to a variable, then you call *reset* or *rewrite* using only the variable.

```
assign (file_variable, 'filename.extension');
reset (file_variable)
```

The method of representing the path differs depending on your operating system. Windows uses backslashes and drive letters due to its DOS heritage (e.g. c:\directory\name.pas), while MacOS X and Linux use forward slashes due to their UNIX heritage.

After you're done with the file, you can close it with:

```
close (File_Identifier);
```

Here's an example of a program that uses files. This program was written for Turbo Pascal and DOS, and will create `file2.txt` with the first character from `file1.txt`:

```
program CopyOneByteFile;

var
```

```
   mychar : char;
   filein, fileout : text;

begin
   assign (filein, 'c:\file1.txt');
   reset (filein);
   assign (fileout, 'c:\file2.txt');
   rewrite (fileout);
   read (filein, mychar);
   write (fileout, mychar);
   close(filein);
   close(fileout)
end.
```

`EOLN` is a Boolean function that is `TRUE` when you have reached the end of a line in an open input file.

```
eoln (file_variable)
```

If you want to test to see if the standard input (the keyboard) is at an end-of-line, simply issue `eoln` without any parameters. This is similar to the way in which *read* and *write* use the console (keyboard and screen) if called without a file parameter.

```
eoln
```

`EOF` is a Boolean function that is `TRUE` when you have reached the end of the file.

```
eof (file_variable)
```

Usually, you don't type the end-of-file character from the keyboard. On DOS/Windows machines, the character is Control-Z. On UNIX/Linux machines, the character is Control-D.

Again find the sum and average of five numbers, but this time read in five integers and display the output in neat columns.

Refer to the original problem specification if needed. You should type in the numbers separated by spaces from the keyboard: `45 7 68 2 34`.

The output should now look like this:

```
Number of integers = 5

Number1:      45
Number2:       7
Number3:      68
Number4:       2
Number5:      34
================
Sum:         156
Average:      31.2
```

As an added exercise, you can try to write the output to a file. However, I won't use files in the problem solution.

```
(* Author:    Tao Yue
   Date:      19 June 1997
   Description:
      Find the sum and average of five predefined numbers
   Version:
      1.0 - original version
      2.0 - read in data from keyboard
*)

program SumAverage;

const
   NumberOfIntegers = 5;

var
   A, B, C, D, E : integer;
   Sum : integer;
   Average : real;

begin    (* Main *)
   write ('Enter the first number: ');
   readln (A);
   write ('Enter the second number: ');
   readln (B);
   write ('Enter the third number: ');
   readln (C);
   write ('Enter the fourth number: ');
   readln (D);
   write ('Enter the fifth number: ');
   readln (E);
   Sum := A + B + C + D + E;
   Average := Sum / 5;
   writeln ('Number of integers = ', NumberOfIntegers);
   writeln;
   writeln ('Number1:', A:8);
   writeln ('Number2:', B:8);
   writeln ('Number3:', C:8);
   writeln ('Number4:', D:8);
   writeln ('Number5:', D:8);
   writeln ('================');
   writeln ('Sum:', Sum:12);
   writeln ('Average:', Average:10:1);
end.
```

Program Flow

Sequential control is simple. The computer executes each statement and goes on to the next statement until it sees an end.

Boolean expressions are used to compare two values and get a true-or-false answer:

```
value1 relational_operator value2
```

The following relational operators are used:

<	less than
>	greater than
=	equal to
<=	less than or equal to
>=	greater than or equal to
<>	not equal to

You can assign Boolean expressions to Boolean variables. Here we assign a true expression to *some_bool*:

```
some_bool := 3 < 5;
```

Complex Boolean expressions are formed by using the Boolean operators:

`not`	negation (~)
`and`	conjunction (^)
`or`	disjunction (v)
`xor`	exclusive-or

NOT is a unary operator — it is applied to only one value and inverts it:

- `not true = false`
- `not false = true`

AND yields `TRUE` only if both values are `TRUE`:

- `TRUE and FALSE = FALSE`
- `TRUE and TRUE = TRUE`

OR yields `TRUE` if at least one value is `TRUE`:

- `TRUE or TRUE = TRUE`
- `TRUE or FALSE = TRUE`
- `FALSE or TRUE = TRUE`
- `FALSE or FALSE = FALSE`

XOR yields `TRUE` if one expression is TRUE and the other is FALSE. Thus:

- `TRUE xor TRUE = FALSE`
- `TRUE xor FALSE = TRUE`
- `FALSE xor TRUE = TRUE`
- `FALSE xor FALSE = FALSE`

When combining two Boolean expressions using relational and Boolean operators, be careful to use parentheses.

```
(3>5) or (650<1)
```

This is because the Boolean operators are higher on the order of operations than the relational operators:

1. not
2. * / div mod and
3. + - or
4. < > <= >= = <>

So `3 > 5 or 650 < 1` becomes evaluated as `3 > (5 or 650) < 1`, which makes no sense, because the Boolean operator `or` only works on Boolean values, not on integers.

The Boolean operators (AND, OR, NOT, XOR) can be used on Boolean variables just as easily as they are used on Boolean expressions.

Whenever possible, don't compare two real values with the equals sign. Small round-off errors may cause two equivalent expressions to differ.

The IF statement allows you to branch based on the result of a Boolean operation. The one-way branch format is:

```
if BooleanExpression then
  StatementIfTrue;
```

If the Boolean expression evaluates to true, the statement executes. Otherwise, it is skipped.

The IF statement accepts only one statement. If you would like to branch to a compound statement, you must use a begin-end to enclose the statements:

```
if BooleanExpression then
begin
  Statement1;
  Statement2
end;
```

There is also a two-way selection:

```
if BooleanExpression then
  StatementIfTrue
else
  StatementIfFalse;
```

If the Boolean expression evaluates to `FALSE`, the statement following the else will be performed. Note that you may *not* use a semicolon after the statement preceding the else. That causes the computer to treat it as a one-way selection, leaving it to wonder where the `else` came from.

If you need multi-way selection, simply nest if statements:

```
if Condition1 then
  Statement1
else
  if Condition2 then
    Statement2
  else
    Statement3;
```

Be careful with nesting. Sometimes the computer won't do what you want it to do:

```
if Condition1 then
  if Condition2 then
    Statement2
else
  Statement1;
```

The else is always matched with the most recent if, so the computer interprets the preceding block of code as:

```
if Condition1 then
  if Condition2 then
    Statement2
  else
    Statement1;
```

You can get by with a null statement:

```
if Condition1 then
  if Condition2 then
    Statement2
  else
else
  Statement1;
```

or you could use a begin-end block. But the best way to clean up the code would be to rewrite the condition.

```
if not Condition1 then
  Statement1
else
  if Condition2 then
    Statement2;
```

This example illustrates where the `not` operator comes in very handy. If Condition1 had been a Boolean like: `(not(a < b) or (c + 3 > 6)) and g`, reversing the expression would be more difficult than NOTting it.

Also notice how important indentation is to convey the logic of program code to a human, but the compiler ignores the indentation.

Suppose you wanted to branch one way if **`b`** is 1, 7, 2037, or 5; and another way if otherwise. You could do it by:

```
if (b = 1) or (b = 7) or (b = 2037) or (b = 5) then
  Statement1
else
  Statement2;
```

But in this case, it would be simpler to list the numbers for which you want Statement1 to execute. You would do this with a `case` statement:

```
case b of
  1,7,2037,5: Statement1;
  otherwise   Statement2
end;
```

The general form of the case statement is:

```
     case selector of
        List1:     Statement1;
        List2:     Statement2;
        ...
        Listn:     Statementn;
        otherwise Statement
     end;
```

The `otherwise` part is optional. When available, it differs from compiler to compiler. In many compilers, you use the word **`else`** instead of `otherwise`.

selector is any variable of an ordinal data type. You may not use reals!

Note that the lists must consist of literal values. That is, you must use constants or hard-coded values -- you cannot use variables.

Looping means repeating a statement or compound statement over and over until some condition is met.

There are three types of loops:

- **fixed repetition** - only repeats a fixed number of times
- **pretest** - tests a Boolean expression, then goes into the loop if TRUE
- **posttest** - executes the loop, then tests the Boolean expression

In Pascal, the fixed repetition loop is the `for` loop. The general form is:

```
     for index := StartingLow to EndingHigh do
        statement;
```

The *index* variable must be of an ordinal data type. You can use the *index* in calculations within the body of the loop, but you should not change the value of the index. An example of using the index is:

```
sum := 0;
for count := 1 to 100 do
  sum := sum + count;
```

The computer would do the sum the long way and still finish it in far less time than it took the mathematician Gauss to do the sum the short way (1+100 = 101. 2+99 = 101. See a pattern? There are 100 numbers, so the pattern repeats 50 times. 101*50 = 5050. This isn't advanced mathematics, its attribution to Gauss is probably apocryphal.).

In the for-to-do loop, the starting value MUST be lower than the ending value, or the loop will never execute! If you want to count down, you should use the `for-downto-do` loop:

```
     for index := StartingHigh downto EndingLow do
        statement;
```

In Pascal, the for loop can only count in increments (steps) of 1.

The pretest loop has the following format:

```
     while BooleanExpression do
        statement;
```

The loop continues to execute until the Boolean expression becomes `FALSE`. In the body of the loop, you must somehow affect the Boolean expression by changing one of the variables used in it. Otherwise, an infinite loop will result:

```
a := 5;
while a < 6 do
  writeln (a);
```

Remedy this situation by changing the variable's value:

```
a := 5;
while a < 6 do
  begin
    writeln (a);
    a := a + 1
  end;
```

The `WHILE ... DO` lop is called a pretest loop because the condition is tested before the body of the loop executes. So if the condition starts out as `FALSE`, the body of the while loop never executes.

The posttest loop has the following format:

```
     repeat
        statement1;
        statement2
     until BooleanExpression;
```

In a `repeat` loop, compound statements are built-in -- you don't need to use `begin`-`end`. Also, the loop continues until the Boolean expression is `TRUE`, whereas the while loop continues until the Boolean expression is `FALSE`.

This loop is called a posttest loop because the condition is tested *after* the body of the loop executes. The `REPEAT` loop is useful when you want the loop to execute at least once, no matter what the starting value of the Boolean expression is.

Problem 1

Find the first 10 numbers in the Fibonacci sequence. The Fibonacci sequence starts with two numbers:

```
1 1
```

Each subsequent number is formed by adding the two numbers before it. 1+1=2, 1+2=3, 2+3=5, etc. This forms the following sequence:

```
1 1 2 3 5 8 13 21 34 55 89 144 ...
```

Problem 2

Display all powers of 2 that are less than 20000. Display the list in a properly formatted manner, with commas between the numbers. Display five numbers per line. The output should look like:

```
     1, 2, 4, 8, 16,
32, 64, 128, 256, 512,
1024, 2048, 4096, 8192, 16384
```

Solution to Fibonacci Sequence Problem

```
(* Author:    Tao Yue
   Date:      19 July 1997
   Description:
      Find the first 10 Fibonacci numbers
   Version:
      1.0 - original version
*)

program Fibonacci;

var
   Fibonacci1, Fibonacci2 : integer;
   temp : integer;
   count : integer;

begin    (* Main *)
   writeln ('First ten Fibonacci numbers are:');
   count := 0;
   Fibonacci1 := 0;
   Fibonacci2 := 1;
   repeat
      write (Fibonacci2:7);
      temp := Fibonacci2;
      Fibonacci2 := Fibonacci1 + Fibonacci2;
      Fibonacci1 := Temp;
      count := count + 1
   until count = 10;
   writeln;

   (* Of course, you could use a FOR loop or a WHILE loop
      to solve this problem. *)

end.     (* Main *)
```

Solution to Powers of Two Problem

```
(* Author:    Tao Yue
   Date:      13 July 2000
   Description:
      Display all powers of two up to 20000, five per line
   Version:
      1.0 - original version
*)

program PowersofTwo;

const
   numperline = 5;
   maxnum = 20000;
   base = 2;

var
   number : longint;
   linecount : integer;

begin    (* Main *)
   writeln ('Powers of ', base, ', 1 <= x <= ', maxnum, ':');
   (* Set up for loop *)
   number := 1;
   linecount := 0;
   (* Loop *)
   while number <= maxnum do
      begin
         linecount := linecount + 1;
         (* Print a comma and space unless this is the first
            number on the line *)
         if linecount > 1 then
            write (', ');
         (* Display the number *)
         write (number);
         (* Print a comma and go to the next line if this is
            the last number on the line UNLESS it is the
            last number of the series *)
         if (linecount = numperline) and not (number * 2 > maxnum) then
            begin
               writeln (',');
               linecount := 0
            end;
         (* Increment number *)
         number := number * base;
      end;  (* while *)
   writeln;

   (* This program can also be written using a
      REPEAT..UNTIL loop. *)

end.     (* Main *)
```

Note that I used three constants: the base, the number of powers to display on each line, and the maximum number. This ensures that the program can be easily adaptable in the future.

Using constants rather than literals is a good programming habit to form. When you write really long programs, you may refer to certain numbers thousands of times. If you hardcoded them into your code, you'd have to search them out. Also, you might use the same value in a different context, so you can't simply do a global Search-and-Replace. Using a constant makes it simpler to expand the program.

Also note that I used the `longint` type for the `number` variable. This is because to fail the test number <= 20000, `number` would have to reach 32768, the next power of two after 16384. This exceeds the range of the integer type: -32768 to 32767. (try it without longint and see what happens)

Subprograms

A procedure is a subprogram. Subprograms help reduce the amount of redundancy in a program. Statements that are executed over and over again but not contained in a loop are often put into subprograms.

Subprograms also facilitate top-down design. Top-down design is the tackling of a program from the most general to the most specific. For example, top down design for going from one room to another starts out as:

- Get out of first room
- Go to second room
- Go into second room

Then it is refined to

- Get out of first room
    - Go to door
    - Open the door
    - Get out of door
    - Close door
- ...

Just going to the door can be refined further:

- Get out of first room
    - Go to door
        - Get out of seat
        - Turn towards door
        - Walk until you almost bump into it

This, of course, can be further refined to say how much exercise should be given to your cardiac myofibrils, and how much adenosine diphosphate should be converted to adenosine triphosphate by fermentation or aerobic respiration. This may seem to be too detailed, but for computer programming, this is, in effect what you have to do. The computer can't understand general statements -- you must be specific.

Main tasks should be contained in procedures, so in the main program, you don't have to worry about the details. This also makes for reusable code. You can just keep your procedures in one file and link that into your program.

A procedure has the same basic format as a program:

```
procedure Name;

const
  (* Constants *)

var
  (* Variables *)
```

```
begin
  (* Statements *)
end;
```

There is a semicolon (not a period) at the end.

To call the procedure from the main program, just use the name, like you would `writeln`.

```
     Name;
```

Procedures are very often used to output data. It's that simple (until the next lesson, of course).

A parameter list can be included as part of the procedure heading. The parameter list allows variable values to be transferred from the main program to the procedure. The new procedure heading is:

```
procedure Name (formal_parameter_list);
```

The parameter list consists of several parameter groups, separated by semicolons:

```
param_group_1; param_group2; ... ; param_groupn
```

Each parameter group has the form:

```
identifier_1, identifier_2, ... , identifier_n : data_type
```

The procedure is called by passing arguments (called the actual parameter list) of the same number and type as the formal parameter list.

```
procedure Name (a, b : integer; c, d : real);
begin
  a := 10;
  b := 2;
  writeln (a, b, c, d)
end;
```

Suppose you called the above procedure from the main program as follows:

```
alpha := 30;
Name (alpha, 3, 4, 5);
```

When you return to the main program, what is the value of `alpha`? 30. Yet, `alpha` was passed to `a`, which was assigned a value of 10. What actually happened was that `a` and `alpha` are totally distinct. The value in the main program was not affected by what happened in the procedure.

This is called **call-by-value**. This passes the ***value*** of a variable to a procedure.

Another way of passing parameters is **call-by-reference**. This creates a link between the formal parameter and the actual parameter. When the formal parameter is modified in the procedure, the actual parameter is likewise modified. Call-by-reference is activated by preceding the parameter group with a `VAR`:

```
VAR identifier1, identifier2, ..., identifiern : datatype;
```

In this case, constants and literals are not allowed to be used as actual parameters because they might be changed in the procedure.

Here's an example which mixes call-by-value and call-by-reference:

```
procedure Name (a, b : integer; VAR c, d : integer);
begin
  c := 3;
  a := 5
end;

begin
  alpha := 1;
  gamma := 50;
  delta := 30;
  Name (alpha, 2, gamma, delta);
end.
```

Immediately after the procedure has been run, `gamma` has the value `3` because `c` was a reference parameter, but `alpha` still is `1` because `a` was a value parameter.

This is a bit confusing. Think of call-by-value as copying a variable, then giving the copy to the procedure. The procedure works on the copy and discards it when it is done. The original variable is unchanged.

Call-by-reference is giving the actual variable to the procedure. The procedure works directly on the variable and returns it to the main program.

In other words, call-by-value is one-way data transfer: main program to procedure. Call-by-reference goes both ways.

Functions work the same way as procedures, but they always return a single value to the main program through its own name:

```
function Name (parameter_list) : return_type;
```

Functions are called in the main program by using them in expressions:

```
a := Name (5) + 3;
```

Be careful not to use the name of the function on the right side of any equation inside the function. That is:

```
function Name : integer;
begin
  Name := 2;
  Name := Name + 1
end.
```

is a no-no. Instead of returning the value 3, as might be expected, this sets up an infinite recursive loop. Name will call Name, which will call Name, which will call Name, etc.

The return value is set by assigning a value to the function identifier.

```
Name := 5;
```

It is generally bad programming form to make use of `VAR` parameters in functions -- functions should return only one value. You certainly don't want the sin function to change your pi radians to 0 radians because they're equivalent -- you just want the answer 0.

Scope refers to where certain variables are visible. You have procedures inside procedures, variables inside procedures, and your job is to try to figure out when each variable can be seen by the procedure.

A global variable is a variable defined in the main program. Any subprogram can see it, use it, and modify it. All subprograms can call themselves, and can call all other subprograms defined before it.

The main point here is: within any block of code (procedure, function, whatever), the only identifiers that are visible are those defined *before* that block and either *in* or *outside of* that block.

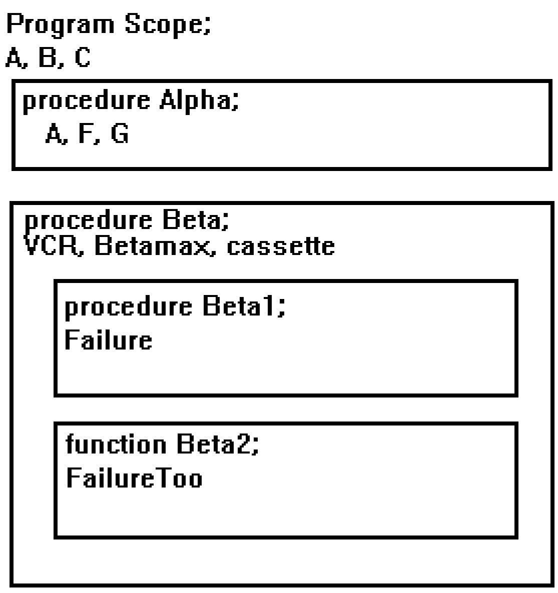
```
program Stupid;
var A;

  procedure StupidToo;
  var A;
  begin
    A := 10;
    writeln (A)
  end;

begin (* Main *)
  A := 20;
  writeln (A);
  StupidToo;
  writerln (A);
end. (* Main *)
```

The output of the above program is:

```
20
10
20
```

The reason is: if two variable with the same identifiers are declared in a subprogram and the main program, the main program sees its own, and the subprogram sees its own (not the main's). The *most local* definition is used when one identifier is defined twice in different places.

Here's a scope chart which basically amounts to an indented copy of the program with just the variables and minus the logic:



- Everybody can see global variables `A, B, and C`.
- However, in procedure `Alpha` the global definition of `A` is replaced by the local definition.
- `Beta1` and `Beta2` can see variables `VCR, Betamax, and cassette`.
- `Beta1` cannot see variable `FailureToo`, and `Beta2` cannot see `Failure`.
- No subprogram except `Alpha` can access `F and G`.
- Procedure `Beta` can call `Alpha` and `Beta`.
- Function `Beta2` can call any subprogram, including itself (the main program is not a subprogram).

Recursion is a difficult topic to grasp. However, it's very easy to apply once you understand it. The programming assignment for this chapter will involve recursion.

Recursion means allowing a function or procedure to call itself. It keeps calling itself until some limit is reached.

The summation function, designated by an uppercase Sigma in mathematics, is a popular example of recursion:

```
function Summation (num : integer) : integer;
begin
  if num = 1 then
    Summation := 1
  else
    Summation := Summation(num-1) + num
end;
```

Suppose you call *Summation* for 3.

```
a := Summation(3);
```

- Summation(3) becomes Summation(2) + 3.
- Summation(2) becomes Summation(1) + 2.
- At 1, the recursion stops and becomes 1.
- Summation(2) becomes 1 + 2 = 3.

- Summation(3) becomes 3 + 3 = 6.
- `a` becomes 6.

Recursion works backward until a given point is reached at which an answer is defined, and then works forward with that definition, solving the other definitions which rely upon that one.

All recursive procedures/functions should have some sort of test so stop the recursion. Under one condition, called the base condition, the recursion should stop. Under all other conditions, the recursion should go deeper. In the example above, the base condition was `if num = 1`. If you don't build in a base condition, the recursion will either not take place at all, or become infinite.

After all these confusing topics, here's something easy.

Remember that procedures/functions can only see variables and other subprograms that have already been defined? Well, there is an exception.

If you have two subprograms, each of which calls the other, you have a dilemma that no matter which you put first, the other still can't be called from the first.

To resolve this chicken-and-the-egg problem, use forward referencing.

```
procedure Later (parameter list); forward;

procedure Sooner (parameter list);
begin
  ...
  Later (parameter list);
end;
...
procedure Later;
begin
  ...
  Sooner (parameter list);
end;
```

The same goes for functions. Just stick a **`forward;`** at the end of the heading.

A classic recursion problem, taught in all introductory Computer Science courses, is the Towers of Hanoi. In this problem, you have three vertical pegs. There is a cone-shaped tower on the leftmost peg, consisting of a series of donut-shaped discs. For example, this is what a four-story tower looks like:

```
   |       |       |
   |       |       |
   *       |       |
  ***      |       |
 *****     |       |
*******    |       |
```

The pegs are designated 1, 2, and 3 from left to right. The challenge is to move a tower (any height) from peg 1 to peg 3. In the process, no large disc may be placed on top of a smaller disc, and only one disc (the topmost disc on a peg) may be moved at any one time.

The problem seems trivial, and it is for one or two discs. For one disc, you simply move it from peg 1 to peg 3. For two discs, move the topmost disc from peg 1 to peg 2, then 1 to 3, and finally move the smaller disc from 2 to 3.

The problem gets harder for three or more discs. For three discs, you'd move 1 to 3, then 1 to 2, then 3 to 2. This effectively creates a two-story tower on peg 2. Then move the largest disc: 1 to 3. Now move the two-story tower on top of the large disc: 2 to 1, 2 to 3, 1 to 3.

Your mission, should you choose to accept it -- write a program using a recursive procedure to solve the Towers of Hanoi for any number of discs. First ask the user for the height of the original tower. Then, print out step-by-step instructions for moving individual discs from one peg to another. For example, a three-disc problem should produce the following output:

```
   1 to 3
   1 to 2
   3 to 2
   1 to 3
   2 to 1
   2 to 3
   1 to 3
```

As stated in the section on recursion (lesson 4E), recursion is one of the more difficult topics to grasp. Some people will look at this problem and find it extremely easy. Others will have a difficult time with it. However, once you get past the hurdle of understanding recursion, the actual coding of the program is relatively simple.

So, if you'd like to challenge yourself, stop reading right here. If you have a little trouble, keep reading for a small hint.

Hint: the problem, like all recursive problems, reduces itself, becoming simpler with each step. Remember the three-disc problem? You first create a two-disc tower on peg 2, which allows you to move the bottommost disc on peg 1 to peg 3. Then you move the two-disc tower on top of peg 3.

It's the same with four discs. First create a three-disc tower on peg 2, then move the biggest disc over to peg 3 and move the three-disc tower to peg 3. How do you create the three-disc tower? Simple. We already know how to move a three-disc tower from peg 1 to peg 3. This time, you're just moving from peg 1 to peg 2, then when the biggest peg is in place, you're moving the tower from peg 2 to peg 3. In this whole procedure, we can act as though the big disc doesn't exist, since it's guaranteed to be bigger than the others and thus poses no problem. Just utilize the three-disc solution, switching the numbers around.

Good luck!

```
(* Author:    Tao Yue
   Date:      13 July 2000
   Description:
      Solves the Towers of Hanoi
   Version:
      1.0 - original version
*)

program TowersofHanoi;

var
   numdiscs : integer;

(********************************************************)

procedure DoTowers (NumDiscs, OrigPeg, NewPeg, TempPeg : integer);
(* Explanation of variables:
      Number of discs -- number of discs on OrigPeg
      OrigPeg -- peg number of the tower
      NewPeg -- peg number to move the tower to
      TempPeg -- peg to use for temporary storage
*)

begin
   (* Take care of the base case -- one disc *)
   if NumDiscs = 1 then
      writeln (OrigPeg, ' ---> ', NewPeg)
   (* Take care of all other cases *)
   else
      begin
         (* First, move all discs except the bottom disc
            to TempPeg, using NewPeg as the temporary peg
            for this transfer *)
         DoTowers (NumDiscs-1, OrigPeg, TempPeg, NewPeg);
         (* Now, move the bottommost disc from OrigPeg
            to NewPeg *)
         writeln (OrigPeg, ' ---> ', NewPeg);
         (* Finally, move the discs which are currently on
            TempPeg to NewPeg, using OrigPeg as the temporary
            peg for this transfer *)
         DoTowers (NumDiscs-1, TempPeg, NewPeg, OrigPeg)
      end
end;

(********************************************************)


begin    (* Main *)
   write ('Please enter the number of discs in the tower ===> ');
   readln (numdiscs);
   writeln;
   DoTowers (numdiscs, 1, 3, 2)
end.     (* Main *)
```

Data Types

You can declare your own ordinal data types. You do this in the `type` section of your program:

```
type
    datatypeidentifier = typespecification;
```

One way to do it is by creating an enumerated type. An enumerated type specification has the syntax:

```
(identifier1, identifier2, ... identifiern)
```

For example, if you wanted to declare the months of the year, you would do a type:

```
type
    MonthType = (January, February, March, April,
          May, June, July, August, September,
          October, November, December);
```

You can then declare a variable:

```
var
    Month : MonthType;
```

You can assign any enumerated value to the variable:

```
Month := January;
```

All the ordinal functions are valid on the enumerated type. `ord(January) = 0`, and `ord(December) = 11`.

A few restrictions apply, though: enumerated types are internal to a program -- they can neither be read from nor written to a text file. You must read data in and convert it to an enumerated type. Also, the idenfier used in the type (such as January) cannot be used in another type.

One purpose of an enumerated type is to allow you, the programmer, to refer to meaningful names for data. In addition, enumerated types allow functions and procedures to be assured of a valid parameter, since only variables of the enumerated type can be passed in and the variable can only have one of the several enumerated values.

A subrange type is defined in terms of another ordinal data type. The type specification is:

```
lowest_value .. highest_value
```

where *lowest_value* **<** *highest_value* and the two values are both in the range of another ordinal data type.

For example, you may want to declare the days of the week as well as the work week:

```
type
    DaysOfWeek = (Sunday, Monday, Tuesday, Wednesday,
                  Thursday, Friday, Saturday);
    DaysOfWorkWeek = Monday..Friday;
```

You can also use subranges for built-in ordinal types such as `char` and `integer`.

Suppose you wanted to read in 5000 integers and do something with them. How would you store the integers?

You could use 5000 variables, lapsing into:

```
     aa, ab, ac, ad, ... aaa, aab, ... aba, ...
```

But this would grow tedious (after declaring those variables, you have to read values into each of those variables).

An array contains several storage spaces, all the same type. You refer to each storage space with the array name and with a subscript. The type definition is:

```
     type
        typename = array [enumerated_type] of another_data_type;
```

The data type can be anything, even another array. Any enumerated type will do. You can specify the enumerated type inside the brackets, or use a predefined enumerated type. In other words,

```
   type
      enum_type = 1..50;
      arraytype = array [enum_type] of integer;
```

is equivalent to

```
   type
      arraytype = array [1..50] of integer;
```

Aside: This is how strings are actually managed internally &mdash as arrays. Back before modern Pascal compilers added native support for strings, programmer had to handle it themselves, by declaring:

```
     type
        String = packed array [0..255] of char;
```

and using some kind of terminating character to signify the end of the string. Most of the time it's the null-character (ordinal number 0, or `ord(0)`). The `packed` specifier means that the array will be squeezed to take up the smallest amount of memory.

Arrays of characters representing strings are often referred to as *buffers*, and errors in handling them in the C or C++ programming languages may lead to *buffer overruns*. A *buffer overrun* occurs when you try to put, say, a 200-character string into a 150-length array. If memory beyond the buffer is overwritten, and if that memory originally contained executable code, then the attacker has just managed to inject arbitrary code into your system. This is what caused the famous Slammer worm that ran rampant on the Internet for several days. Try it in Pascal and see what happens.

Arrays are useful if you want to store large quantities of data for later use in the program. They work especially well with for loops, because the index can be used as the subscript. To read in 50 numbers, assuming the following definitions:

```
     type
         arraytype = array[1..50] of integer;

     var
         myarray : arraytype;
```

use:

```
     for count := 1 to 50 do
         read (myarray[count]);
```

Brackets [ ] enclose the subscript when referring to arrays.

```
     myarray[5] := 6;
```

You can have arrays in multiple dimensions:

```
     type
        datatype = array [enum_type1, enum_type2] of datatype;
```

The comma separates the dimensions, and referring to the array would be done with:

```
     a [5, 3]
```

Two-dimensional arrays are useful for programming board games. A tic tac toe board could have these type and variable declarations:

```
     type
         StatusType = (X, O, Blank);
         BoardType = array[1..3,1..3] of StatusType;
     var
         Board : BoardType;
```

You could initialize the board with:

```
     for count1 := 1 to 3 do
         for count2 := 1 to 3 do
             Board[count1, count2] := Blank;
```

You can, of course, use three- or higher-dimensional arrays.

A record allows you to keep related data items in one structure. If you want information about a person, you may want to know name, age, city, state, and zip.

To declare a record, you'd use:

```
     TYPE
        TypeName = record
```

```
           identifierlist1 : datatype1;
           ...
           identifierlistn : datatypen;
        end;
```

For example:

```
     type
         InfoType = record
             Name : string;
             Age : integer;
             City, State : String;
             Zip : integer;
         end;
```

Each of the identifiers Name, Age, City, State, and Zip are referred to as fields. You access a field within a variable by:

```
     VariableIdentifier.FieldIdentifier
```

A period separates the variable and the field name.

There's a very useful statement for dealing with records. If you are going to be using one record variable for a long time and don't feel like typing the variable name over and over, you can strip off the variable name and use only field identifiers. You do this by:

```
     WITH RecordVariable DO
        BEGIN
           ...
        END;
```

Example:

```
     with Info do
         begin
             Age := 18;
             ZIP := 90210;
         end;
```

A pointer is a data type which holds a memory address. A pointer can be thought of as a reference to that memory address, while a variable accesses that memory address directly. If a variable is someone's phone number, then a pointer is the page and line number where it's listed in the phone book. To access the data stored at that memory address, you ***dereference*** the pointer.

To declare a pointer data type, you must specify what it will point to. That data type is preceded with a carat (^). For example, if you are creating a pointer to an integer, you would use this code:

```
     type
        PointerType = ^integer;
```

You can then, of course, declare variables to be of type PointerType.

Before accessing a pointer, you block off an area in memory for that pointer to access. This is done with:

```
     New (PointerVariable);
```

To access the data at the pointer's memory location, you add a carat after the variable name. For example, if `PointerVariable` was declared as type `PointerType` (from above), you can assign the memory location a value by using:

```
     PointerVariable^ := 5;
```

After you are done with the pointer, you must deallocate the memory space. Otherwise, each time the program is run, it will allocate more and more memory until your computer has no more. To deallocate the memory, you use the **`Dispose`** command:

```
     Dispose(PointerVariable);
```

A pointer can be assigned to another pointer. However, note that since only the ***address***, not the value, is being copied, once you modify the data located at one pointer, the other pointer, when dereferenced, also yields modified data. Also, if you free (or deallocate) a pointer, the copied pointer now points to meaningless data.

What is a pointer good for? Why can't you just use an integer in the examples above instead of a pointer to an integer? Well, the above is clearly a contrived example. The real power of pointers is that, in conjunction with records, it makes dynamically-sized data structures possible. If you need to store many items of one data type in order, you can use an array. However, your array has a predefined size. If you don't have a large enough size, you may not be able to accomodate all the data. If you have a huge array, you take up a lot of memory when sometimes that memory is not being used.

A dynamic data structure, on the other hand, takes up only as much memory as is being used. What you do is to create a data type that points to a record. Then, the record has that pointer type as one of its fields. For example, stacks and queues can all be implemented using this data structure:

```
     type
         PointerType = ^RecordType;
         RecordType = record
             data : integer;
         next : PointerType;
     end;
```

Each element points to the next. The last record in the chain indicates that there is no next record by setting its `next` field to a value of **`nil`**.

Final words

This concludes my informal course in Pascal. You should have a reasonable understanding of the basics of Pascal by now, though this tutorial is by no means comprehensive.

Compiler manuals contain a wealth of information about additional syntactical elements and predefined routines for use in your programs. Here's the Free Pascal Compiler manual. The Reference guide contains information about the system unit which is used automatically with the program, while the units reference manual gives an idea of how many routines there are for accomplishing all sorts of tasks.

Good luck in your future Pascal endeavors! And when you move on to other languages and find yourself using PascalCasing for variable names, remember how Pascal has left an indelible mark on computer programming even though other languages have risen to prominence.