Pascal Tutorial

TABLE OF CONTENTS

INTRODUCTION

The Pascal programming language is named after the great French mathmatician, Blaise Pascal. It was originally tought as a tutorial language in colleges, and was not designed to be a popular commercial compiler. It's popularity was at first a surprise, but it has grown to become one of the most popular and widely-known languages.

For this tutorial, it may be helpful to know that I am using the Turbo Pascal version 7.0 compiler. You can find it in stores for about $120.00, but you might be able to find it cheaper used. It also may be informative that I will not go into in-depth detail on the Pascal language. I will probably only cover the basics of scope and procedure calls, syntax, and BGI graphics. But it will be enough to get you programming in Pascal on your own.

SECTION 1 - THE PASCAL PROGRAMMING STRUCTURE

The syntax of the pascal programming language is remarkably simple, but it is very different from other programming languages. The first line in a Pascal program is traditionally the name of the program. Here is an example of naming the program:

```
PROGRAM Test_Program_1;
```

This gives the current program a name that the compiler can use and recognize. At this stage it may be importaant to notice the semicolon atthe end of the line. Semicolons are used in pascal to signify the end of a whole command and/or line.. We'll look at them in more depth later.

Preprocessor directives follow the program name in a Pascal program. The Pascal compiler does not know what functions you want to use in the program, so you must include a unit in your program that contains the function declarations. For example, if you want to use input and output functions, you include a unit called crt, which stands for cathode ray tube. If you want to use BGI (Borland Graphics Interface) graphics, you must include the graphics unit. Pascal units are included with the Uses statement at the head of a program. Here is an example of a program which might use the cathode ray tube and the BGI graphics functions:

```
USES CRT, Graphics;
```

Memorizing which units must be included in the preprocessor directives at the begining of a program is sometimes a difficult task, but most of what you will need is found within these two units.

Pascal, like most other programming languages, use variables to store data. At this stage, all of our variables will be global variables. A global variable is variable that can be accessed from anywhere in the program, and is declared in a var block, usually at the head of the program. In Pascal, you must declare variables before you can use them, or else the compiler will not recoginze it. In a var block, the name of the variable comes first, followed by a colon, and the data type. Note that whitespaces (blank spaces, or tabs) are not read by the compiler, so use them wisely to improve readability! Here is an example of a common global var block:

```
VAR
  UserName : String;
  Score    : Integer;
  Average  : Real;
```

Pascal has a great deal of variable types that you can use in your programs. Here's a table of the various data types supported in Pascal

TYPE	RANGE	USE
Shortint	-128 to 127	storing small whole numbers
Integer	-32768 to 32767	storing medium-sized whole numbers
Longint	-2147483648 to 2147483647	storing large whole numbers
Byte	0 to 255	represents a single byte of memory
Word	0 to 65535	represents a word of memory (2 bytes)
real	2.9e-39 to 1.7e38	storing numbers with digits after the decimal point (non-integer)
single	1.5e-45 to 3.4e38	storing numbers with digits after the decimal point (non-integer)
double	5.0e-324 to 1.7e308	storing large numbers with digits

		after the decimal point
extended	3.4e-4932 to 1.1e4932	storing enormous numbers with digits after the decimal point
comp	-9.2e18 to 9.2e18	storing 64-bit whole numbers
char	0 to 255	holds a character, such as a letter, number,or symbol
string	0	holds multiple characters, forming textual info like names

Every Pascal program must start somewhere, so you must assign a starting point with a main procedure. The main procedure is a procedure which designates the begining and end of the program: the begining of the main procedure is the begining of the program, and the end of the main procedure is the end of the program (unless you end it prematurely). I can't spell. The writers of Pascal decided to use simple to understand words to designate the begin and end of a procedure: Begin and End. Amazing. Heres how it works:

```
BEGIN
  { main procedure code goes here }
END.
```

There are a couple distinguishing features of this. First, the main procedure in a Pascal program does not have any identification... it is identified only by the words Begin and End. The code for the main procedure goes within these keywords. Secondly, the braces represent remark statements in Pascal. You can make a remark several lines long.... all you have to do is put a brace before the remark and a brace after the remark. Notice the period after the End. This signifies that this is the end of the program. Once this statement is encountered, the program exectution is done. So here is a template of what a minimal Pascal program may look like:

```
{ Program name }
PROGRAM qp7_program_1;

{ Preprocessor directives }
USES crt;

{ Global Variable Block }
VAR
  Int_var    : Integer;
  String_var : String;
  Any_var    : Real;

{ Main Procedure }
BEGIN
  { Main procedure code }
```

```
END.
```

All of the programs we will write in this tutorial are in this basic form. Comments in Pascal are enclosed in {} braces. Any text inside of the {} braces are comments, and have no affect on the compilation of the program. The comments are not termintated by a carriage return, so you must put the } endbrace to terminate the comment. This is nice because comments can be several lines long without added complexity.

So as you can see, Pascal is a highly structured language and is actually very simple due to this organization. Now that you know how a Pascal program is set up, you're ready to start programming!

SECTION 2 - THE BASIC FUNCTIONS

When dealing with text in Pascal, there are a few very simple functions which are usefull in making text input / output programs. These functions recieve input from the user and send output to the screen. To get a text string from the user, use the readln function with the variable to store the inputted value in as the argument. To display a line of text to the screen, use the writeln function with the text value as the argument.

```
Program user_input;
USES crt;
VAR
  text : string;
BEGIN
  writeln( 'What is your name?' );
  readln ( text );
  writeln;
  writeln( 'Hello ', text );
  readln;
END.
```

First, we display a prompt on the screen for the user to input their name. Then we read from the keyboard until the unser presses enter and store the value into the string text. Then we print a blank line. Then we print a warm message to the user using two arguments. When two arguments are used in a writeln function, both are printed to the screen. If the argument is enclosed in " marks, it echos that to the screen. If it is not enclosed in " marks, it prints the value of the variable. Finnally, we have the readln which waits until the user pushes the return key and ends the program.

There are two main functions used for outputting data to the screen: write and writeln. We have alread seen writeln used, and write is virtually the same. The only difference is that writeln prints the arguments and then adds a carriage return character so the cursor goes to the next line after the function is executed. Here are two examples:

```
PROGRAM write_statements;
USES crt;
VAR
  string1 :   string;
  string2 :   string;

BEGIN
  string1 := 'hello there ';
  string2 := 'how are you?';
  writeln (string1);
  writeln (string2);
  write (string1);
  write (string2);
  readln;
END.
```

Notice the assignment in the first two lines of the main procedure. When you assign a value to a variable, the = opperator must be proceded by a :. Forgeting the colon is my most common error in Pascal. Ok, the output of this program looks like this:

```
hello there
how are you?
hello there how are you?
```

The writeln function returns to the next line after the text has been printed, but the write function keeps the current cursor position so the next time a function is called the text is added at the same location. BASIC users may note similarity between BASIC's PRINT statement and Pascal's writeln.

Sometimes you need to prompt the user to input only one key, and then immediately continue with the code. The BASIC equivalent of the readkey function is the structure DO: LOOP WHILE INKEY$ = "". Program execution is halted until the user presses a key, and the text value of that key is returned. So lets set up a quick program

```
PROGRAM get_key;
USES crt;
```

```
VAR
  inkey : string;

BEGIN
  clrscr;
  writeln ('Do you like fish? [yn]');
  inkey := readkey;
  if inkey = 'y' then
  begin
    writeln('Good.  I find fish delicious.');
  end
  else
  begin
    writeln('I think rainbows are pretty.');
  end;
END.
```

I've introduced a lot here. Ok, first the screen is cleared and the user is prompted to give his opinion on fish. The [yn] is standard dos for "push y or n." The value of the key they push is stored into the varaible inkey. Then we have a conditional testing structure. If the value of inkey is y, (the user pushed the y key) then print something, but if the user pushed anything else, then print something else. Notice the exact structure of the conditional test. This is the extended method, and using the begin and end keywords you can place several lines of code which will be performed in result of a conditional test. You need the ; after the last end so the compiler knows you are done with the conditional test.

The write and writeln functions also provide convenient methods for formating a numeric value. You can define the minimum number of characters a number will take up, and you can limit the number of digits after the decimal place. Look at the following code snippet:

```
writeln ('Input a number with some decimal places');
readln (amt);
writeln ('Your number in dollars is $', amt:0:2 );
```

The :0:2 after the variable to be printed specifies how the variable's value will be displayed. In this case, there is no minimum number of digits (0) and there can be up to 2 digits after the decimal point (2).

These functions make Pascal a very simple yet useful language for such applications as databases and text programs. But there is more, as you will soon see.

SECTION 3 - SPECIAL FUNCTIONS AND CONSTRUCTS

Every programming languages has its own way of doing common programming tasks, and compared to other languages, the way Pascal does things is quite simple. However, learning any programming language involves memorizing the constructs of the language, which are the way things are set up. Pascal is used primarily as an instructional language because these constructs are make logical sense are are easy to memorize. Every programming language has is own syntax for loops, converting case, setting ASCII codes, and other things, and learning the syntactical part of the language is simply a matter of learning these constructs. Pascal is one of the more less-awkward languages in this regard. In fact, Pascal is really like a low-level C because it can do most of the same things, but the method of doing them is far easier to understand. This is why it was (and is) used in schools to introduce students to programming.

Loops are simple, and they are pretty much the same as in BASIC. You can identify a loop by the keyword do at the definition. Unlike BASIC loops, which start with a DO and end with a LOOP, or start with a FOR and end with a NEXT, Pascal loops define the condition or durration of the loop and then define the body using the familiar begin and end keywords. One thing to remember about for loops is that an assignment is involved, so you have to use the := opperator. Here is an example of two kinds of loops:

```
PROGRAM loops_demo;
USES crt;
VAR
  id : integer;

BEGIN
  clrscr;
  for id := 1 to 10 do
  begin
    write( id, ' ' );
  end;
  writeln;
  while id <= 10 do
  begin
    id := id + 1;
    write( id, ' ' );
  end;
  readln;
END.
```

This program simply prints the numbers from 1 to 10 twice, but it does it different ways. The first method is a for loop which increases id by 1 until it equals 10. The next loop is a while loop, which increases id while it is less than or equal to 10. This yields the same results.

Arrays in Pascal are different from BASIC arrays, and are slightly more awkward to declare. Recall that an array is a variable which can have more than one value called data members, or elements. It involves a variable declaration with the array keyword, the bounds within brackets, and the type of which the array is of. Here is an example of a small program implementing an array

```
PROGRAM array_demo;
USES crt;
VAR
  numbers : array[1 .. 10] of integer;
  i       : integer;

BEGIN
  clrscr;
  { assign data to array }
  for i := 1 to 10 do
  begin
    numbers[i] := i * 10;
  end;

  { print data in array }
  for i := 1 to 10 do
  begin
    write( numbers[i], ' ' );
  end;
  readln;
END.
```

So first we declare the array. Notice the array keyword, and then the bounds inside the brackets. The lower bound is on the left, then come two periods, and then the upper bound is on the right. After the bracket we define what type the array will be of. Arrays can only be of one type. In the program we assign values to the array with a loop, and then print them with another, just to show that it works. There are a number of other functions which can come in usefull in Pascal programming. I'll go through a few of them.

length: returns the length in characters of a string given as the arugment. It is fairly straigtforward. If a string has 5 characters in it, the number 5 is returned. Heres a quick code example:

```
PROGRAM length_demo;
USES crt;
VAR
  text    : string;
  textlen : integer;

BEGIN
  clrscr;
  writeln ('What is you name?');
  readln (text);
  textlen := length(text);
  writeln ('Your name is ', textln, ' characters long.');
  readln;
END.
```

upcase: the upcase function converts a single character to the equivalent upper case ASCII character. This is unlike the UCASE$ BASIC function because it cannot be used directly on strings. But what if you want to convert a whole string to upper case? Well, a string is actually an array of characters, so that tells you that you can loop through the string, treating it as an array, and re-assign the uppercase value. Here's how you would do it:

```
PROGRAM convert_upper;
USES crt;
VAR
  text : string;
  i    : integer;

BEGIN
  clrscr;
  writeln ('Input some text in all lowercase letters');
  readln ('text');
  for i := 1 to length(text) do
  begin
    text[i] := upcase(text[i]);
  end;
  writeln ('The uppercase text is: ');
  writeln ('  ', text);
  readln;
END.
```

So we loop for the number of characters in the string and convert each character to the equivalent uppercase character one at a time. We store the uppercase character into the old string, so the string is completely re-written in the uppercase. Then we print the string.

chr: this function prints the ASCII symbol for a number given. The ASCII character set is a set of 256 text symbols and screen opperations that have numeric values. The ASCII value for the letter 'a' is 65, for example. ASCII characters 0-32 are special because they mostly do not print anything on the screen, but have some other opperation. ASCII character 13 is the carriage return, which makes the cursor go to the next line. It is the equivalent of pushing the return key on your keyboard. The Pascal chr function returns the symbol when you pass a number. It is the equivalent of the BASIC function CHR$. Lets print out the ASCII characters from 32 to 255.

```
PROGRAM print_asciis;
USES crt;
VAR
  i,ii : integer;

BEGIN
  clrscr;
  writeln ('ASCII Characters 32-255');
  i := 32;
  while i <= 255 do
  begin
    for ii := 0 to 9 do write (i+ii,'=',chr(i+ii),'  ');
    i := i + 10;
    writeln;
  end;
  readln;
END.
```

This looks a bit weird, but it works. First we assign 32 to i, which is our starting point. Then we loop until i is 255 or greater. The inner loop prints one line of information... 9 character codes. First we print the ordinal number which represents the character, then we print the character. We increase i by 10, go to the next line, and print the next 9 characters on the next line. If you try to run this program, take notice of all the weird characters 219 and 178. These are commonly used in games for text graphics. This is one method of making big text, although Pascal provides ways of enlarging text through BGI graphics.

ord: This is the opposite of chr. When you pass an ASCII character, it returns the ASCII code for that character. So if you pass an 'A' to the ord function, it will return 97. I won't

bother to go into a huge explanation on this. It's pretty simple. It is the equivalent of the BASIC ASC function.

There are quite a few more, but it is redundant to repeat synonymns for BASIC commands. You'll find that if you choose to go further into Pascal that it is a lot like BASIC. However, it is supperior to BASIC in the way that it is faster, it is more dynamic, and you can program efficient code in your own style. This is a feature which has made C++ my language of choice, but we'll get to that!

SECTION 4- PROCEDURES

Procedures make code more efficient by placing blocks of code which can be reused in it's own section. You can also divide a program up into procedures to make it easier to read and easier to edit. Imagine programming a game as a team in GW-BASIC which has no procedures to speak of. Several programmers must find their place in a mass of lines and line numbers, and edit other people's code they may be unfamiliar with. Now imagine a team of programmers programming completely seperate parts of the game. One person codes a procedure that handles the keyboard and joystick input. Another codes the graphics engine, and yet another does the screen interface. With high level languages you can divide code into procedures, each of which might serve its own purpose or to assist another sub with its purpose.

A procedure is not executed until it is called. When the program starts, it calls the main procedure. The main procedure can then call other procedures, and so on. There are two parts to a procedure in Pascal: the definition and the call. The procedure should be defined before the main procedure definition, which starts with a BEGIN and ends with an END.. A procedure, on the other hand, starts with the procedure name, and then the definition between the BEGIN and END keywords.

Values called arguments can also be passed to a procedure in what is called an argument list. The act of taking a value and giving it to a procedure is called passing an argument. Recall that when a procedure is defined, the name of the procedure comes first. After the procedure name comes the arugment list, which is the values which were passed to the procedure which can be used in it. These values come from where it was called, and are passed in the call statement... but we'll get to that. Lets go through the parts of a procedure because you are probably not quite clear on this:

```
PROGRAM proc_demo;
USES crt;
VAR
  number:  integer;
  addnum:  integer;
```

```
{ Add procedure }
PROCEDURE add( arg: integer );
BEGIN
  number := number + arg;
END;

{ Main procedure }
BEGIN
  clrscr;
  write('Input a number: ');
  readln(number);
  write('Input a number to add to that: ');
  readln(addnum);
  add(addnum);
  writeln('The number is now', number);
  readln;
END.
```

Here is the classic example of a trivial program... that is, it serves no usefull purpose but to demonstrate a use. I've made a procedure called add which adds a certain value to a certain number. These certain values are given by the user. So here's how it works. The user is asked for a number. Then the user is asked for another number. The add procedure is called, which adds the two together and stores the value in the first number. Then the new number is printed out. Pretty easy.

Now say we want to improve on this a bit. Say we want to pass a value to a procedure, and have the procedure give us a value. Before, we changed the value of a global variable from within the procedure, but there is a more efficient way to do this: functions. Functions can be passed a value from where it was called, and can return a value to where it was called. We specifiy a return value by setting the value we want to return equal to the name of the function. Well, that would mean that the function would have to have a data type, since it is being treated as a variable. So we must specifiy the return value of the function, which goes after the argument list (where we identify the values passed to the function). So lets make a program that allows us to input a number in degrees centigrade and have it convert that number to degrees kelvin. We'll call the convert procedure from the main procedure, where the user inputs the data.

```
PROGRAM convert_temps;
USES crt;
VAR
  degree: real;

{ Convert function: returns a real degree kelvin
```

```
    based on a real degree centigrade passed to it. }
FUNCTION convert( degC: real ): real
BEGIN
  convert := degC + 273.15;
END;

{ Main procedure }
BEGIN
  clrscr;
  write('Input a number in degrees centigrade: ');
  readln(degree);
  writeln('The equivalent is ', convert(degree), ' degrees kelvin');
  readln;
END.
```

If you don't know what degrees kelvin are, don't worry about it. All you need to know is that 0° centigrade is equal to 273° kelvin. Why, well because the lowest possible temperature is absolute zero, and that is -273° centigrade or 0° kelvin, but now out of Chemistry and back to programming! Ok, so we declare the convert function to accept one real argument which will be local to that procedure, and the return type is real. Notice we use the FUNCTION keyword instead of PROCEDURE so we can specify that there will be a return value. In the function declaration itself, notice we assign the value we want to return to where we called it the name of the function. In the main procedure, we can treat it as a normal variable because once the compiler comes across the name of the function, it is executed and an expression can be evaluated for the value returned from the function. The convert(degree) expression actually becomes degree + 273.15 because the function return value is substituted in where the function was called.

In my QBasic tutorial I touched on a subject called scope. When the program encounters a variable declaration, and memory is allocated for that variable. The durration of time memory is allocated for is determined by where in the code the variable is declared. If you declare a variable outside any procedure, then that variable has a global scope, and memory is allocated when the program starts and exists for the durration of the program. So if you give a value to a global variable and don't manually change it, it will have the same value when the program ends. Furthermore, any procedure or function can access the value of a global variable because it's scope is just that; global. But there is another type of variable scope called local scope. Local scope is where memory for a variable is allocated at the beginning of a procedure, and that memory is released when the procedure ends. If you try to access that variable from outside the procedure, you will get an "undefined variable" error. Only the procedure that declared the variable can read the value of the local variable. Local variables are declared at the beginning of a procedure in it's own var block. Here's what an entire program using procedures and scope rules might look like:

```
PROGRAM scope_demo;
```

```
USES crt;
VAR
  { global variable declarations }

PROCEDURE proc1(arg1, arg2 : integer);
VAR
  { local variable declarations }
BEGIN
  { code for procedure }
END;

FUNCTION proc2(arg1, arg2 : integer) : integer;
VAR
  { local variable declarations }
BEGIN
  { code for procedure }
END;

BEGIN
  { code for main procedure }
END.
```

Whatever is declared in the local variable blocks is local to that procedure, meaning it can only be accessed from within that procedure and memory is not used up in the global program. The local variables of proc1 and proc2 could even be the same name, but they cannot read each other's. The main procedure can only use global variables, so we should try to solve that problem. Global variables add what is called overhead to a program. When a variable declaration is encountered in a global variable declaration block, memory is allocated and remains allocated until the program ends. It requires a certain ammount of memory (RAM) for the variables to be allocated, and this is overhead. So we can solve this by converting all global variables into local ones. If we use efficient methods, we can avoid name-mangling and reduce the total overhead. But since the main procedure has no form of local variables, it must use global variables. So therefore, any variables declared global that are only used for the purpose of the main procedure is wasted overhead. It would be much more efficient to write a procedure to handle the main code and let the main procedure call this one. In some cases, as for small programs, it may be better just to use global variables like we have so far, but it is better practice to use a calling procedure. I wont always do this in my tutorail for simplicity reasons, but if you plan to have a great deal of code in the main procedure, it would be better to put it all in a seperate one.

SECTION 5 - BGI GRAPHICS

Throughout this tutorial I have mentioned numerous times how similar Pascal is to BASIC. Graphics in Pascal are one exception. It is harder to use the graphics functions but they produce a much wider array of effects.

The first thing you must learn about graphics in Pascal is that you have to initialize the graphics mode before you can use them. In BASIC, you have to initialize the screen mode with the SCREEN statement, and you're doing the same thing in Pascal only differently. There are three components to initializing the graphics: graphics driver, graphics mode, and the location of the graphics libraries. Finding the graphics driver on your computer can usually be done by using the DETECT value. You don't need to worry about assigning a value to the graphics mode, and the location of the graphics libaries will be in the bgi directory of the compiler you are using. For example, on my computer the bgi directory is in the c:\tp\bgi directory. You usually want to initialize the graphics in the main procedure so you can use graphics in any other procedure, but you can also intialize the graphics in only the procedure you want to use graphics in. I will usually initgraph in the main procedure for simplicity sake. Here's an example of how this process is done:

```
PROGRAM init_graph;
USES graph, crt;
VAR
  grdriver, grmode:  integer;

BEGIN
  grdriver := DETECT;
  initgraph(grdriver, grmode, 'c:\tp\bgi');
  readln;
END.
```

This program won't do anything, but it is kind of cool to know that you just initialized the graphics mode. Actually its not that cool, but who cares. Notice that I had to include the graph header file at the top in the uses block. This is because you are using the bgi functions library which is contained in the graph header file, so wherener you do graphics you need to include that. Now lets do a program that actually does something.

There are 79 Bgi functions supported by Pascal. Bgi stands for Borland Graphics Interface. Using the bgi libary is a lot like calling graphics functions in BASIC, only usually the functions are a bit more complex. I'm not going to go into what specifically each function does since if you have a Pascal compiler you can just look in the help index. One usefull thing my version of Pascal does not have is a list of all the bgi functions. However, by C++ compiler does, so I've printed out the list here. If you want to know what a specific function does, just look it up in your Pascal help index. If any of these don't work in Pascal, email me. There's not much use for some of them in every day programming, so some might only apply to C++.

arcbar
bar3d
circle

cleardevice
clearviewport
closegraph
detectgraph
drawpoly
ellipse
fillellipse
fillpoly
floodfill
getarccoords
getaspectratio
getbkcolor
getcolor
getdefaultpalette
getdrivername
getfillpattern
getfillsettings
getgraphmode
getimage
getlinesettings
getmaxcolor
getmaxmode
getmaxx
getmaxy
getmodename
getmoderange
getpalette
getpalettesize
getpixel
gettextsettings
getviewsettings
getx
gety
graphdefaults
grapherrormsg
graphresult
imagesize
initgraph
installuserdriver
installuserfont
line
linerel
lineto
moverel
moveto
outtext
outtextxy

pieslice
putimage
putpixel
rectangle
registerbgidriver
registerbgifont
restorecrtmode
sector
setactivepage
setallpalette
setaspectratio
setbkcolor
setcolor
setfillpattern
setfillstyle
setgraphbufsize
setgraphmode
setlinestyle
setpalette
setrgbpalette
settextjustify
settextstyle
setusercharsize
setviewport
setvisualpage
setwritemode
textheight
textwidth

Let's try a couple of these out in a program so you can see how they work. They're pretty straightforward, so you should be able to understand this program pretty easily:

```
PROGRAM bgi_demo;
USES graph, crt;
VAR
  grdriver, grmode: integer;

BEGIN
  grdriver := DETECT;
  initgraph(grdriver, grmode, 'c:\tp\bgi');

  setcolor(9);
  line(0,0,getmaxx,getmaxy);
  setfillstyle(1,4);
  bar(100,100,200,200);
```

```
  readln;
END.
```

Wow, real impressive. You can look up any of the functions you need to in the help index. It is very extensive, and will help you learn the graphics part of Pascal a lot quicker than I could tell you. Plus it would be futile to print out information on every function in Pascal when the exact same thing would be in the help index. But I hope this section gave you a feel of how simple graphics in Pascal can be. It is only a matter of learning and memorizing the many functions in the bgi libary before you can make anything graphically.

SECTION 6 - PROGRAMMING STYLES

In Pascal, you are introduced to the concept of procedural programming. This is key if you are to program in Visual Basic or C, or you plan to work on a program with other people in a team. You may not recognize it at first, but programming has a style. You can tell one person's style by the way they program, the