Topic: 2.4.2 Program testing

2.4.2 Program testing

Types of programming Errors

There are several types of errors, with consequences ranging from deficiencies in the formatting of the output to the calculation of wrong results. A ***compilation error*** (which prevents the compiler from compiling the source code) is usually a *syntax error* but could be an error in the compiler itself. A ***syntax error*** results when the source code does not obey the rules of the language. The compiler generates error messages to help the programmer to fix the code. The source code may compile to machine code which then fails upon execution. A ***run-time error*** causes this situation. Potentially the most serious type of error occurs when the program appears to be working but is performing faulty processing due to ***logic errors*** in the source code. We classify the common errors as syntax errors, run-time errors, and logic errors, but begin with three very common errors for beginners that do not fit neatly into any of these categories.

Common Syntax Errors

Do not be discouraged if your first program generates many error messages. Often there are *cascading errors*, in which a single mistake such as a misspelled identifier in a variable declaration gives rise to several error messages. You then make one correction and several messages disappear at once. In the following table showing extracts from error messages to look out for, note how some messages are much more helpful than others.

Topic: 2.4.2 Program testing

Description of error	Typical message content in Delphi	Typical message content in Lazarus	Comment
Misspelled identifier	Undeclared identifier	Identifier not found	Spelling of identifier must be consistent
Semicolon immediately before else	';' not allowed before 'ELSE'	**";" expected but "ELSE" found**	The **if** … **then** … **else** is one statement and should not have a semicolon immediately before**else**.
Use of := instead of = in test of equality e.g. **if** Total := 10 **then** ...	Type of expression must be BOOLEAN	"THEN" expected but ":=" found	
Identifier of variable contains space e.g. **var** my num : **real**;	',' or ':' expected but …	":" expected but …	
Identifier of variable begins with digit e.g. **var** 2ndPlace:**string**;	identifier/declaration expected but number found	BEGIN" expected but "ordinal const" found	Identifier must begin with letter or underscore
Identifier of procedure begins with digit e.g. **procedure** 5Times;	identifier expected but number found	"identifier" expected but "ordinal const" found	Identifier must begin with letter or underscore
Use of double quotes instead of single quotes to enclose a string literal e.g. Surname := "Coe";	Illegal character in input file ($22)	Illegal character ($22)	
String literal not enclosed by quotes e.g. Forename := Joe;	Undeclared identifier	Identifier not found	
Use of round brackets to enclose array index e.g.MyArray(63)	Missing operator or semicolon	";" expected but "(" found	Use square brackets to enclose array index.
Real number starts with decimal point e.g. MyReal := .34;	Expression expected but '.' found	Illegal expression	Include the leading 0 e.g. MyReal := 0.34;

Topic: 2.4.2 Program testing

Argument type does not match parameter type when calling a subroutine	Incompatible types	Incompatible types	
Expression operated on by logical operators not enclosed in brackets. e.g.**until** Count = MAX **or** Found;	Operator not applicable to this operand type	Incompatible types	Should be: **until** (Count = MAX) **or** Found;

Description	Example	Comment
Missing type check, allowing wrong type of data to be entered	Letter "a" entered and assigned to integer variable	Validate entered data.

Index of array not within declared array bounds	Example of INCORRECT code:
	
	programArrayBounds;
	{$APPTYPE CONSOLE}{$R+}
	uses
	SysUtils;
	const
	Sales : **array**[1..4] **ofinteger** =
	(2435, 3423, 3410, 2865);
	var
	Index : **integer**;
	begin
	write('View sales for which quarter? ' +
	'1, 2, 3 or 4 ');
	readln(Index);
	writeln(Sales[Index]);
	readln;
	end.
		If the user hits a wrong key (e.g. 5 instead of 4) there could be a run-time error.
		
		The compiler directive {$R+} enables run-time checking of ranges. With the default directive {$R-}, the effects of the user's mistake are difficult to predict.
		
		Include validation code to accept only the digits 1 to 4.

Overflow e.g. when a number is too large to be held in the type of variable used	Example of INCORRECT code: **program** Overflow;	The compiler directive {$Q+} enables run-time checking of overflow. With the default {$Q-}, the program runs

Topic: 2.4.2 Program testing

	{$APPTYPE CONSOLE}{$Q+}
	uses
	SysUtils;
	var
	MillisecsPerDay, Jan_ms :
	integer;
	{Use int64 instead of integer
	to make
	this program work.}
	begin
	MillisecsPerDay := 24 * 60 * 60
	* 1000;
	
	Jan_ms := MillisecsPerDay *
	31;
	writeln(Jan_ms);
	readln;
	end.
		and outputs a negative number of milliseconds in January! An integer is held in two's complement format, which represents a negative number when the most significant bit is 1.

Topic: 2.4.2 Program testing

Common Run-time errors

Some errors cannot be detected by the compiler and only become apparent at run time. For example, errors which depend on the input from a user cannot be predicted by the compiler.

Division by 0	Example of INCORRECT code:
	
	programDivisionByZero;
	{$APPTYPE CONSOLE}
	uses
	SysUtils;
	var
	Distance, Time, Speed : **real**;
	begin
	write('Distance in m? ');
	readln(Distance);
	write('Time in s? ');
	readln(Time);
	Speed := Distance / Time;
	write('Speed in m/s: ');
	writeln(Speed : 6 : 2);
	readln;
	end.
		An entry of 0 for Time will cause a run-time error when Speed is calculated. The program needs validation to ensure that Time is greater than zero. (It also needs type checking, perhaps using the val procedure, to accept only valid numbers
		for Distance and Time).

Infinite loop	Example of INCORRECT code: Num := 0; **repeat** Num := Num + 3; **until**Num = 10;	The integer Num "misses" the end of the loop, which continues to run. Use safe conditions such as **until**Num>= 10 in this example.

Topic: 2.4.2 Program testing

Common logic errors

When you have removed all the syntax errors, compile-time errors and run-time errors you may think it is time to heave a huge sigh of relief - but don't forget that there may still be logical errors lurking in your program.

Out-by-one (or off-by-one) errors are very common, hence the emphasis on boundary testing in computing courses. The cause of the error is often the operator used, as in the first example below.

Description	Example	Comment
Out-by-one	Example of INCORRECT code: **if** Marks[i] >PassMark**then** inc(NumberOfPasses);	The >= operator should have been used instead of >.
Out-by-one	Example of INCORRECT code: NumCoaches := NumPeople**DIV**SeatingCapacity;	The calculated number of coaches may be one less than required.
Missing **begin** and **end** to group statements intended to be run together depending on some condition	Example of INCORRECT code: **if** Names[i] = SearchName**then** writeln('Found at position ', i); Found := True;	This code will assign True to Foundwhether or not the condition is true. This error is often made when adding an instruction to existing code. If you always use **begin** and **end** after **then**,**else** and **do**, your programs will be safeguarded from this type of error.
Semicolon immediately after **then**,**else** or **do**	Example of INCORRECT code: **for**i := 1 **to** 10 **do**; writeln('Line ', i);	In this code the **for** loop has no code to execute other than incrementing i. After the loop, the writeln statement is executed once using the final value of i.
Operator precedence not what is required to give the right answer	Example of INCORRECT code: Celsius := Fahrenheit - 32 * 5 / 9	The operators * and / have precedence over + and -. In order to perform the subtraction first, the expression Fahrenheit - 32 should be enclosed in brackets. **Tip:** If you are unsure of the operator precedence, use brackets to force the order of calculation to be the one you require.

Topic: 2.4.2 Program testing

How can I avoid making mistakes while coding any program?

Follow these tips to reduce the number of errors in your programs:

- Take frequent breaks; mistakes are more likely when you are tired.
- If in doubt, check. Look at examples if you are unsure of the correct syntax. Read the checklists above from time to time and be on the lookout for those mistakes.
- Learn from your mistakes. Try to recognize the types of mistake you make most often and concentrate on reducing those.
- Do not rely on the compiler to find your mistakes; it will only find certain types of mistake.
- Check the computer output carefully to make sure it is correct. Your tests should include normal, boundary and erroneous data. See our sample test plan in the tutorial on testing.
- Include checks in the program itself, such as validating the input and intermediate results.
- Try your hand at spotting the errors in our selection of Programs to Debug. See how many of our deliberate errors you are able to spot and fix.