Study Guide

embarcadero

GETTING STARTED WITH WINDOWS AND MAC DEVELOPMENT

Programming Language Basics

Delphi and C++

E-Learning Series Course Book

Lesson 4

Embarcadero Technologies

© Copyright 2012 Embarcadero Technologies, Inc. All Rights Reserved.

Americas Headquarters
100 California Street, 12th Floor
San Francisco, California 94111

EMEA Headquarters
York House
18 York Road
Maidenhead, Berkshire
SL6 1SF, United Kingdom

Asia-Pacific Headquarters
L7. 313 La Trobe Street
Melbourne VIC 3000
Australia

Lesson 4 – Programming Language Basics – Delphi and C++

Version: 1.2
Presented: May 24, 2012
Last Updated: June 20, 2012
Prepared by: David Intersimone "David I", Embarcadero Technologies
© Copyright 2012 Embarcadero Technologies, Inc. All Rights Reserved.
davidi@embarcadero.com
http://blogs.embarcadero.com/davidi/

Contents

Lesson 4 – Programming Language Basics – Delphi and C++ 2
Introduction 5
Object Oriented Programming (OOP) 6
Objects 7
Properties, Methods and Events 7
Properties 8
Methods 8
Events 9
Private, Protected, Public and Published 9
Constructors and Destructors 9
Language Guides 10
Fundamental Elements 10
Delphi Fundamental Elements 10
Keywords 10
Directives 11
Identifiers 11
Numbers 12
Character strings 12
Labels 13
Comments and Compiler Directives 13
C and C++ Fundamental Elements 13
Keywords 13
Identifiers 14

Decimal Constants 14
Octal Constants 15
Hexadecimal Constants 15
Floating Point Constants 15
Enumeration Constants 15
Character Constants 15
String Literals 16
Labels 17
Comments 17
Data Types 18
Delphi Data Types 18
Integer data types 18
Floating point data types 19
String and Character Data Types 19
File Data Types 19
Boolean Data Types 19
Other Data Types 19
C++ Data Types 20
Integer Data Types 20
Boolean Data Types 20
Floating Point Data Types 20
Character Data Types (String is a class in C++) 20
Other Data Types 20
Statements 21
Delphi Statements 21
C++ Statements 23
Functions (Delphi/C++) and Procedures (Delphi) 26
Delphi Procedures and Functions 26
Procedure Declarations 26
Function Declarations 27
C++ Functions 27
Operators 30

Delphi Operators 30
Binary Arithmetic Operators 30
Unary arithmetic operators 30
Boolean Operators 30
Logical (Bitwise) Operators 31
String operators 31
Pointer Operators 31
Set Operators 32
Relational Operators 33
The @ Operator 33
Operator Precedence 34
C++ Operators 34
Assignment Operators 34
Arithmetic operators 35
Bitwise Operators 35
Logical operators 36
Comma Operator 36
Relational Operators 37
Equality Operators 38
Conditional Operators 38
Unary Operators 39
Plus and Minus Operators 39
Reference/Dereference Operators 40
sizeof Operator 40
typeid Operator 41
Array Subscript Operator 41
. Direct Member Selection Operator 42
-> Indirect Member Selection Operator 42
Increment / Decrement Operators 42
Primary Expression Operators 43
Precedence of Operators 44
Program Structure of a FireMonkey Application 45

Example “FireMonkey HD Application – Delphi” project contents 45
The Main Program source code file – Project1.pas 45
The Main Delphi Application Form – form view and text view 46
The Main Delphi Application Form – Unit Source code 47
Example “FireMonkey HD Application – C++Builder” project” contents 48
The Main C++ Project Source Code – Project6.cpp 48
The Main C++ Application Form – form view and text view 52
The Main C++ Application Form – Unit Source Code and Header File 52
Using Forms 53
FMX.Forms.TForm 54
Exploring one of the Delphi and C++ example programs – ControlsDemo 54
Object Design (using the UML Class Diagram) 55
Handling Windows and Mac Platform Specific Code 56
Conditional Compilation 56
TOSVersion 57
Summary, Looking Forward, To Do Items, Resources, Q&A and the Quiz 57
To Do Items 58
Recommended Reading – Books and eBooks 58
Delphi 58
C++ 59
Links to Additional Resources 59
Delphi: 59
C++: 59
Q&A: 60
Self Check Quiz 64
Answers to the Self Check Quiz: 64

Introduction

You can develop your FireMonkey applications in either “RAD” C++ or Delphi, two easy to learn component based object oriented programming languages that are used by millions of developers worldwide.

Delphi is an elegant and easy to learn object oriented language. Based on Object Pascal, its benefits include easy-to-read code, quick compilation, and the use of multiple unit files for modular programming. Delphi has special features that support the RAD Studio component framework and environment.

You can also develop FireMonkey applications in C++, a widely-adopted, object-oriented programming language. C++Builder's RAD environment allows developers to visually design user interfaces and connect to data and services, yet code in ANSI/ISO compatible C and C++. Developers get the power of C++ with the productivity of rapid development.

In lesson 4 we will take a look at how the Delphi and C++ programming languages work with the FireMonkey business application platform to help you build Windows and Mac applications. This lesson will not cover the complete language specification as there are plenty of books, online tutorials and videos that cover the modern capabilities of Delphi and C++.

Delphi and C++ language guides are available on the Embarcadero DocWiki at:

- http://docwiki.embarcadero.com/RADStudio/XE2/en/Delphi_Language_Guide_Index
- http://docwiki.embarcadero.com/RADStudio/XE2/en/C%2B%2B_Language_Guide_Index

At the end of this lesson course book I have included additional links to sources of information about the Delphi and C++ languages.

Object Oriented Programming (OOP)

Object-oriented programming is an extension of structured programming that emphasizes code reuse and encapsulation of data with functionality. Once you define a class, you and other programmers can use it in different applications, thus reducing development time and increasing productivity.

A class is a data type that encapsulates data and operations on data in a single unit. Before object-oriented programming, data and operations (functions) were treated as separate elements. An object is an instance of a class. That is, it is a value whose type is a class. The term object is often used more loosely in this documentation and where the distinction between a class and an instance of the class is not important, the term "object" may also refer to a class.

You can begin to understand objects if you understand Pascal records or structures in C. Records are made of up fields that contain data, where each field has its own type. Records make it easy to refer to a collection of varied data elements.

Objects are also collections of data elements. But objects, unlike records or structures, contain procedures and functions that operate on their data. These procedures and functions are called methods.

The combination of data and functionality in a single unit is called encapsulation. In addition to encapsulation, object-oriented programming is characterized by inheritance and polymorphism.

Inheritance means that objects derive functionality from other objects (called ancestors); objects can modify their inherited behavior.

Polymorphism means that different objects derived from the same ancestor support the same method and property interfaces, which often can be called interchangeably. Polymorphism in object oriented programming also can provide a way for different objects to use the same methods but implement them in different ways. In the real world, for example, you can ask an object to “speak” but depending on the implementation of the ancestors, the “speak” method might talk like a human, bark like a dog, send Morse code through a telegraph, emit Whale sounds underwater or send JSON packets via HTTP.

Scope determines the accessibility of an object's fields, properties, and methods. All members declared in a class are available to that class and often to its descendants. Although a method's implementation code appears outside of the class declaration, the method is still within the scope of the class because it is declared in the class declaration.

Objects

An object is a specific region of memory that can hold a fixed or variable value (or set of values). (This use of the word object is different from the more general term used in object-oriented languages.) Each value has an associated name and type (also known as a data type). The name is used to access the object. This name can be a simple identifier, or it can be a complex expression that uniquely references the object. The type is used:

- to determine the correct memory allocation requirements.
- to interpret the bit patterns found in the object during subsequent accesses.
- in many type-checking situations, to ensure that illegal assignments are trapped.

Declarations establish the necessary mapping between identifiers and objects. Each declaration associates an identifier with a data type. Most declarations, known as defining declarations, also establish the creation (where and when) of the object; that is, the allocation of physical memory and its possible initialization. Other declarations, known as referencing declarations, simply make their identifiers and types known to the compiler. There can be many referencing declarations for the same identifier, especially in a multi-file program, but only one defining declaration for that identifier is allowed.

Generally speaking, an identifier cannot be legally used in a program before its declaration point in the source code. Legal exceptions to this rule (known as forward references) are labels, calls to undeclared functions, and class, struct, or union tags.

Properties, Methods and Events

In the component based world of FireMonkey, the classes are based on properties, methods, and events. Each class includes data members (properties), functions that operate on the data (methods), and a way to interact with users of the class (events).

Properties

Properties are characteristics of an object that influence either the visible behavior or the operations of the object. For example, the **Visible** property determines whether an object can be seen in an application interface. Well-designed properties make your components easier for others to use and easier for you to maintain.

Here are some of the useful features of properties:

- Unlike methods, which are only available at run time, you can see and change some properties at design time and get immediate feedback as the components change in the IDE.
- You can access some properties in the Object Inspector, where you can modify the values of your object visually. Setting properties at design time is easier than writing code and makes your code easier to maintain.
- Because the data is encapsulated, it is protected and private to the actual object.
- The calls to get and set the values of properties can be methods, so special processing can be done that is invisible to the user of the object. For example, data could reside in a table, but could appear as a normal data member to the programmer.
- You can implement logic that triggers events or modifies other data during the access of a property. For example, changing the value of one property may require you to modify another. You can change the methods created for the property.
- Properties can be virtual.
- A property is not restricted to a single object. Changing one property on one object can affect several objects. For example, setting the Checked property on a radio button affects all of the radio buttons in the group.

Methods

Methods define the behavior of an object. A method is a procedure or function that is associated with a class. A method can access any member, with any visibility, in the same class. Within a class declaration, methods appear as procedure and function headings, which work like forward declarations. Somewhere after the class declaration, but within the same module, each method must be implemented by a defining declaration.

Within the implementation of a method, the identifier Self (Delphi) or this (C++) references the object in which the method is called.

The methods declared in a class can be classified in 3 categories, presented in the following table.

Category	Delphi syntax	C++ syntax	Significance of Self (Delphi) or this (C++)
regular	N/A	N/A	Class instance
static	static	static	N/A
class	class	__classmethod	Meta-class instance

Events

An event is an action or occurrence detected by a program. Most modern applications are said to be event-driven, because they are designed to respond to events. In a program, the programmer has no way of predicting the exact sequence of actions a user will perform. For example, the user may choose a menu item, click a button, or mark some text. You can write code to handle the events in which you are interested, rather than writing code that always executes in the same restricted order.

Regardless of how an event is triggered, FireMonkey objects look to see if you have written any code to handle that event. If you have, that code is executed; otherwise, the default event handling behavior takes place.

Private, Protected, Public and Published

A class type declaration contains three or four possible sections that control the accessibility of its fields and methods:

- The public section declares fields and methods with no access restrictions. Class instances and descendent classes can access these fields and methods. A public member is accessible from wherever the class it belongs to is accessible - that is, from the unit where the class is declared and from any unit that uses that unit.
- The protected section includes fields and methods with some access restrictions. A protected member is accessible within the unit where its class is declared and by any descendent class, regardless of the descendent class's unit.
- The private section declares fields and methods that have rigorous access restrictions. A private member is accessible only within the unit where it is declared. Private members are often used in a class to implement other (public or published) methods and properties.
- For classes that descend from TPersistent, a published section declares properties and events that are available at design time. A published member has the same visibility as a public member, but the compiler generates runtime type information for published members. Published properties appear in the Object Inspector at design time.

When you declare a field, property, or method, the new member is added to one of these four sections, which gives it its visibility: private, protected, public, or published.

Constructors and Destructors

A constructor is a special method that creates and initializes instance objects. A destructor is a special method that destroys the object where it is called and de-allocates its memory.

You can read more about Delphi constructors and destructors on the Embarcadero DocWiki at
http://docwiki.embarcadero.com/RADStudio/en/Methods#Constructors

You can read more about C++ constructors and destructors on the Embarcadero DocWiki at
http://docwiki.embarcadero.com/RADStudio/en/Introduction_To_Constructors_And_Destructors

Language Guides

This section gives a quick overview of the elements of the Delphi and C++ languages including information about fundamental elements, data types, statements, functions, procedures and operators.

Fundamental Elements

Fundamental elements in the Delphi and C++ programming languages include identifiers, numbers, character strings, labels and comments. Let's take a look at each one for Delphi and C++.

Delphi Fundamental Elements

Fundamental language elements for Delphi include keywords (sometimes called reserved words), directives, identifiers, numbers, character strings, labels, comments and compiler directives.

Keywords

The following keywords (reserved words) cannot be redefined or used as identifiers:

```
and, array, as, asm,
begin,
case, class, const, constructor,
destructor, dispinterface, div, do, downto,
else, end, except, exports,
file, finalization, finally, for, function,
goto,
if, implementation, in, inherited, initialization,
inline, interface, is,
label, library,
mod,
nil, not,
object, of, or,
packed, procedure, program, property,
raise, record, repeat, resourcestring,
set, shl, shr, string,
then, threadvar, to, try, type,
unit, until, uses,
var,
while, with,
xor
```

Note: In addition to the words in the preceding table, private, protected, public, published, and automated act as reserved words within class type declarations, but are otherwise treated as directives. The words at and on also have special meanings, and should be treated as reserved words. The keywords of object are used to define method pointers.

Directives

Delphi has more than one type of directive. One meaning for "directive' is a word that is sensitive in specific locations within source code. This type of directive has special meaning in the Delphi language, but, unlike a reserved word, appears only in contexts where user-defined identifiers cannot occur. Hence -- although it is inadvisable to do so -- you can define an identifier that looks exactly like a directive.

```
absolute, abstract, assembler, automated,
cdecl, contains (7),
default, delayed (11), deprecated, dispid, dynamic,
experimental, export, external,
far (1) ,final, forward,
helper (8),
implements, index, inline (2),
library (3), local (4),
message,
name, near (1), nodefault,
operator (10), out, overload, override (7),
package, pascal, platform,
private, protected, public, published,
read, readonly, reference (9), register,
reintroduce, requires (7), resident (1),
safecall,sealed (5), static, stdcall, strict, stored
unsafe
varargs, virtual
winapi (6), write, writeonly
```

Notes: 1. far, near, and resident are obsolete. 2. inline is used directive-style at the end of procedure and function declaration to mark the procedure or function for inlining, but became a reserved word for Turbo Pascal. 3. library is also a keyword when used as the first token in project source code; it indicates a DLL target. Otherwise, it marks a symbol so that it produces a library warning when used. 4. local was a Kylix directive and is ignored for Delphi for Win32. 5. sealed is a class directive with odd syntax: 'class sealed'. A sealed class cannot be extended or derived (like final in C++). 6. winapi is the same as stdcall for Delphi for Win32; 64-bit is different. 7. package, when used as the first token, indicates a package target and enables package syntax. requires and contains are directives only in package syntax. 8. helper indicates "class helper for." 9. reference indicates a reference to a function or procedure. 10. operator indicates class operator. 11. The delayed directive is described in Libraries and Packages.

Identifiers

- Identifiers denote constants, variables, fields, types, properties, procedures, functions, programs, units, libraries, and packages.
- An identifier can be of any length, but only the first 255 characters are significant. An identifier must begin with an alphabetic character, a Unicode character, or an underscore (_) and cannot contain spaces.
- Alphanumeric characters, Unicode characters, digits, and underscores are allowed after the first character.
- Reserved words cannot be used as identifiers.
- Since the Delphi Language is case-insensitive, an identifier like CalculateValue could be written in any of these ways.

Numbers

- Integer and real constants can be represented in decimal notation as sequences of digits without commas or spaces, and prefixed with the + or - operator to indicate sign. Values default to positive (so that, for example, 67258 is equivalent to +67258) and must be within the range of the largest predefined real or integer type.
- Numerals with decimal points or exponents denote reals, while other numerals denote integers. When the character E or e occurs within a real, it means "times ten to the power of". For example, 7E2 means 7 * 10^2, and 12.25e+6 and 12.25e6 both mean 12.25 * 10^6.
- The dollar-sign prefix indicates a hexadecimal numeral, for example, $8F. Hexadecimal numbers without a preceding - unary operator are taken to be positive values. During an assignment, if a hexadecimal value lies outside the range of the receiving type an error is raised, except in the case of the Integer (32-bit integer) where a warning is raised. In this case, values exceeding the positive range for Integer are taken to be negative numbers in a manner consistent with 2's complement integer representation.

Character strings

- A character string, also called a string literal or string constant, consists of a quoted string, a control string, or a combination of quoted and control strings. Separators can occur only within quoted strings.
- A quoted string is a sequence of characters, from an ANSI or multibyte character set, written on one line and enclosed by apostrophes. A quoted string with nothing between the apostrophes is a null string. Two sequential apostrophes in a quoted string denote a single character, namely an apostrophe.
- The string is represented internally as a Unicode string encoded as UTF-16. Characters in the Basic Multilingual Plane (BMP) take 2 bytes, and characters not in the BMP require 4 bytes.
- A control string is a sequence of one or more control characters, each of which consists of the # symbol followed by an unsigned integer constant from 0 to 65,535 (decimal) or from $0 to $FFFF (hexadecimal) in UTF-16 encoding, and denotes the character corresponding to a specified code value. Each integer is represented internally by 2 bytes in the string. This is useful for representing control characters and multi-byte characters.

Labels

- You can use either an identifier or a non-negative integer number as a label. The Delphi compiler allows numeric labels from 0 to 4294967295 (uint32 range).
- Labels are used in goto statements.

Comments and Compiler Directives

Comments are ignored by the compiler, except when they function as separators (delimiting adjacent tokens) or compiler directives. There are several ways to construct comments:

- { Text between left and right braces is a comment. }
- (* Text between left-parenthesis-plus-asterisk and an asterisk-plus-right-parenthesis is also a comment *)
- // Text between double-slash and end of line is a comment.

Comments that are alike cannot be nested. For instance, (*{}*) will. This latter form is useful for commenting out sections of code that also contain comments.

Here are some recommendations about how and when to use the three types of comment characters:

- Use the double-slash (//) for commenting out temporary changes made during development. You can use the Code Editor's convenient CTRL+/ (slash) mechanism to quickly insert the double-slash comment character while you are working.
- Use the parenthesis-star "(*...*)" both for development comments and for commenting out a block of code that contains other comments. This comment character permits multiple lines of source, including other types of comments, to be removed from consideration by the compiler.
- Use the braces ({}) for in-source documentation that you intend to remain with the code.
- A comment that contains a dollar sign ($) immediately after the opening { or (* is a compiler directive. For example, {$WARNINGS OFF} tells the compiler not to generate warning messages.

A complete list of compiler directives can be found on the Embarcader DocWiki at
http://docwiki.embarcadero.com/RADStudio/XE2/en/Delphi_Compiler_Directives_(List)_Index

C and C++ Fundamental Elements

Fundamental language elements for C and C++ include keywords (sometimes called reserved words), identifiers, decimal constants, octal constants, hexadecimal constants, floating point constants, enumeration constants, character constants, string literals, labels and comments.

Keywords

You can find an alphabetical listing of C and C++ keywords on the Embarcadero DocWiki at http://docwiki.embarcadero.com/RADStudio/en/Keywords,_Alphabetical_Listing_Index. You can also find an index of keywords by category on the Embarcadero DocWiki at http://docwiki.embarcadero.com/RADStudio/en/Keywords,_By_Category_Index

Identifiers

Identifiers are arbitrary names of any length given to classes, objects, functions, variables, user-defined data types, and so on. Identifiers can contain any characters (including ASCII and Unicode), the underscore character "_", and the digits 0 to 9. All characters in an identifier are significant.

- The first character of an identifier must be a letter or an underscore.
- By default, the compiler recognizes all characters as significant. The number of significant characters can be reduced by menu and command-line options, but not increased. To change the significant character length, set the length you want in **Project > Options > C++ Compiler > Advanced > Identifier Length**.

Unicode characters in an identifier universal-character-name, must have an encoding in ISO 10646 that falls into one of the ranges specified in Annex A of TR 10176:2003. Ranges are specified for Latin, Greek, Cyrillic, Armenian, Hebrew, Arabic, Syriac, Thaana, Devanagari, Bengali, anGurmukhi, Gujarati, Oriya, Tamil, Telugu, Kannada, Malayalam, Sinhala, Thai, Lao, Tibetan, Myanmar, Georgian, Ethiopic, Cherokee, Canadian Aboriginal syllabics, Ogham, Runic, Khmer, Mongolian, Hiragana, Katakana, Bopomofo, CJK Unified Ideographs, Yi, Hangul, digits, and special characters.

Identifiers in C and C++ are case sensitive, so that Sum, sum and suM are distinct identifiers.

Global identifiers imported from other modules follow the same naming and significance rules as normal identifiers. However, you have the option of suspending case sensitivity to allow compatibility when linking with case-insensitive languages. With the case-insensitive option, the globals Sum and sum are considered identical, resulting in a possible. "Duplicate symbol" warning during linking.

An exception to these rules is that identifiers of type __pascal are always converted to all uppercase for linking purposes.

Although identifier names are arbitrary (within the rules stated), errors result if the same name is used for more than one identifier within the same scope and sharing the same name space. Duplicate names are legal for different name spaces regardless of scope rules.

Decimal Constants

Decimal constants from 0 to 4,294,967,295 are allowed. Constants exceeding this limit are truncated. Decimal constants must not use an initial zero. An integer constant that has an initial zero is interpreted as an octal constant. Thus,

- int i = 10; /*decimal 10 */
- int i = 010; /*decimal 8 */
- int i = 0;   /*decimal 0 = octal 0 */

Octal Constants

All constants with an initial zero are taken to be octal. If an octal constant contains the illegal digits 8 or 9, an error is reported. Octal constants exceeding 037777777777 are truncated.

Hexadecimal Constants

All constants starting with 0x (or 0X) are taken to be hexadecimal. Hexadecimal constants exceeding 0xFFFFFFFF are truncated.

Floating Point Constants

A floating-point constant consists of:

- Decimal integer
- Decimal point
- Decimal fraction
- e or E and a signed integer exponent (optional)
- Type suffix: f or F or l or L (optional)

You can omit either the decimal integer or the decimal fraction (but not both). You can omit either the decimal point or the letter e (or E) and the signed integer exponent (but not both). These rules allow for conventional and scientific (exponent) notations.

Negative floating constants are taken as positive constants with the unary operator minus (-) prefixed.

Enumeration Constants

Enumeration constants are identifiers defined in enum type declarations. The identifiers are usually chosen as mnemonics to assist legibility. Enumeration constants are integer data types. They can be used in any expression where integer constants are valid. The identifiers used must be unique within the scope of the enum declaration. Negative initializers are allowed. See Enumerations and enum (keyword) for a detailed look at enum declarations.

The values acquired by enumeration constants depend on the format of the enumeration declaration and the presence of optional initializers.

Character Constants

A character constant is one or more characters enclosed in single quotes, such as 'A', '+', or '\n'. In C, single-character constants have data type int. In C++, a character constant has type char. Multi-character constants in both C and C++ have data type int.

String Literals

String constants, also known as string literals, form a special category of constants used to handle fixed sequences of characters. A string literal is of data type array-of- const char and storage class static, written as a sequence of any number of characters surrounded by double quotes:

- "This is literally a string!"
- The null (empty) string is written "".
- The characters inside the double quotes can include escape sequences. This code, for example:
- "\t\t\"Name\"\\\tAddress\n\n"
    - \t provides a tab character
    - \n provides a new line character
    - The \" provides interior double quotes.
    - If you compile with the -A option for ANSI compatibility, the escape character sequence "\\" is translated to "\" by the compiler.

A literal string is stored internally as the given sequence of characters plus a final null character ('\0'). A null string is stored as a single '\0' character.

By default, string literals are ANSI strings containing char characters. You can use the L, u, and U prefixes, before string literals, to specify that string literals should contain wide-characters or Unicode characters (Unicode Character Types and Literals (C++0x)):

A string literal preceded immediately by an L is a wide-character string containing characters of the wchar_t data type. When wchar_t is used in a C program, it is a type defined in stddef.h header file. In C++ programs, wchar_t is a keyword. The memory allocation for wchar_t strings is two bytes per character. The value of a single wide-character is that character's encoding in the execution wide-character set.

In C++ 2011 programs, a string literal preceded immediately by an u character is a Unicode-character string containing characters of the char16_t data type. In C++2011 programs, char16_t is a keyword declaring a 16-bit character type. char16_t defines UTF-16 character encoding for Unicode. The memory allocation for char16_t characters is two or four bytes per character.

In C++2011 programs, a string literal preceded immediately by an U character is a Unicode-character string containing characters of the char32_t data type. In C++2011 programs, char32_t is a keyword declaring a 32-bit character type. char32_t defines UTF-32 character encoding for Unicode. The memory allocation for char32_t characters is four bytes per character.

That is, in C++2011 programs, we can use the following four types of string literals:

- "ANSI string" - this is an ANSI string literal containing char characters;
- L"Wide-character string" - this string literal contains wchar_t characters;
- u"UTF-16 string" - this string literal contains char16_t Unicode characters in UTF-16 encoding;
- U"UTF-32 string" - this string literal contains char32_t Unicode characters in UTF-32 encoding;

Labels

Labels are used in a labeled statement. Labels follow the same rules as Identifiers.

A statement can be labeled in two ways:

```
label-identifier : statement
```

The label identifier serves as a target for the unconditional goto statement. Label identifiers have their own name space and have function scope. In C++ you can label both declaration and non-declaration statements.

```
case constant-expression : statement
                 default : statement
```

case and default labeled statements are used only in conjunction with switch statements.

Comments

Comments are pieces of text used to annotate a program. Comments are for the programmer's use only; they are stripped from the source text before parsing. There are two ways to delineate comments: the C method and the C++ method. The compiler supports both methods, and provides an additional, optional extension permitting nested comments. If you are not compiling for ANSI compatibility, you can use any of these kinds of comments in both C and C++ programs.

C comments

A C comment is any sequence of characters placed after the symbol pair /*. The comment terminates at the first occurrence of the pair */ following the initial /*. The entire sequence, including the four comment-delimiter symbols, is replaced by one space after macro expansion. Note that some C implementations remove comments without space replacements.

C++ comments

C++ allows a single-line comment using two adjacent slashes (//). The comment can start in any position, and extends until the next new line:

```
class X {  // this is a comment
```

```
... };
```

You can also use // to create comments in C code. This feature is specific to the C++Builder compiler and is generally not portable.

Nested comments

ANSI C doesn't allow nested comments. The attempt to comment out a line

```
/*  int /* declaration */ i /* counter */; */
```

fails, because the scope of the first /* ends at the first */. This gives

```
i ; */
```

which would generate a syntax error.

To allow nested comments, check **Project>Options>Advanced Compiler>Source>Nested Comments**.

Data Types

In addition to objects, Delphi and C++ languages include support for standard data types. You can use these data types for local, global variables and members within objects.

Delphi Data Types

Delphi data types include the integer data types, floating point data types, string and character data types, field data types, Boolean data types and a collection of other data types.

Integer data types

- Byte: 8-bit unsigned integer
- ShortInt: 8-bit signed integer
- Word: 16-bit unsigned integer
- SmallInt: 16-bit signed integer
- Cardinal: 32-bit unsigned integer
- LongWord: 32-bit unsigned integer
- Integer: 32-bit signed integer
- LongInt: 32-bit signed integer
- UInt64: 64-bit unsigned integer
- Int64: 64-bit signed integer
- NativeUInt: 64-bit or 32-bit platform-dependent unsigned integer

- NativeInt: 64-bit or 32-bit platform-dependent signed integer

Floating point data types

- Single: Single precision floating-point value (4 bytes)
- Double: Double precision floating-point value (8 bytes)
- Extended: Extended precision floating-point value (10 bytes on Win32, but 8 bytes on Win64)
- Real: alias of Double

String and Character Data Types

- AnsiChar: ANSI character
- Char: Wide character (16-bit)
- WideChar: 16-bit character
- AnsiString: Represents a dynamically allocated ANSI string whose maximum length is limited only by available memory.
- RawByteString: Use as a "codepage-agnostic" parameter to a method or function, or as a variable type to store BLOB data.
- UnicodeString: Unicode string
- String: Alias for UnicodeString
- ShortString: A string of maximum 255 characters
- WideString: A string of 16-bit characters

File Data Types

- File: File descriptor
- TextFile, Text: Text file descriptor

Boolean Data Types

- Boolean: Represents a logical value (true or false)
- ByteBool: Represents an 8-bit logical value.
- WordBool: Represents a 16-bit logical value.
- LongBool: Represents a 32-bit logical value.

Other Data Types

- Enumeration: has a value from a defined range of possible values (TSuitofCards = (Clubs, Diamonds, Hearts, Spades)
- Subrange: defines a sub range of possible values (0..9, ‘A’..’Z’)
- Set: contains a bitmap of values (up to 255) from a list of possible values (TCharacters = set of [‘A’..’Z’, ‘a’..’z’];)
- Array: Represents an indexed collection of elements of the same type.

- Record: Represents a heterogeneous set of elements.
- Variant: Represents values that can change type at run time.
- Pointer: Represents a pointer to data of any type.
- Currency: A fixed-point data type used to hold monetary values.

C++ Data Types

C++ data types include integer data types, boolean data types, floating point data types, character data types and a collection of other data types.

Integer Data Types

- int – 32 bit signed integer
- unsigned – 32 bit unsigned integer
- short, short int, short signed int - 16 bit signed integer
- unsigned short, unsigned short int – 16 bit unsigned integer
- byte – 8 bits
- word – 16 bits
- __int64 – 64 bit signed integer
- unsigned __int64 – 64 bit unsigned integer

Boolean Data Types

- bool – Represents a logical value (true or false) – 8 bits (top 7 bits are ignored)

Floating Point Data Types

- float – single precision floating point (4 bytes)
- double – double precision floating point (8 bytes)

Character Data Types (String is a class in C++)

- char - character (size is platform specific)
- signed char - signed character (size is platform specific)
- unsigned char - unsigned character
- wchar_t - wide character (16 bits)
- char16_t - 16 bit UTF-16 Unicode character (C++ 2011)
- char32_t - 32-bit UTF-32 Unicode character (C++ 2011)

Other Data Types

- void * - pointer
- int * - pointer to integer
- char * - pointer to character
- float * - pointer to floating point
- int identifier [n] – array of n integers
- struct – data structure containing a group of members
- union, anomymous unions – define the same memory as different data types
- typedef – used to define your own data types
- enum – enumerations of values – type compatible with numeric variables

Statements

Statements in Delphi and C++ include assignment and compound statements, the if and case (Delphi) and switch (C++) statements. Loop statements include for (Delphi and C++), repeat (Delphi), while (Delphi and C++) and do while (C++) statements. This section will cover each statement type for both languages.

Delphi Statements

Assignment statement: assign a variable or expression to another variable

```
variable := variable or expression
```

Compound statement: used to contain a block of several statements.

```
begin
   <statement(s)>
end
```

If statement: conditional test and execution of one or more statements.

```
if <logical expression> then
  <statement or compound statement>

if <local expression> then
  <statement or compound statement>
else
  <statement or compound statement>
```

Case statement: provides a list of expression values and statements to execute for each value. Sometimes the case statement is easier to read and follow than a series of nested if/then/else statements.

```
case expression of
  caselist1: <statement or compound statement>
```

```
  …
  caselistn: <statement or compound statement>
else
  <statement or compound statement>
end
```

For statement: loop through statements based on a range of values or a collection or enumeration of values.

```
for <counter> := <initial value> to <final value>
  do <statement or compound statement>

for <counter> := <initial value> downto <final value>
  do <statement or compound statement>

for <element> in <some collection or enumeration>
  do <statement or compound statement>
```

Repeat statement: repeating loop with the logical test at the end.

```
repeat
  <statement of block of statements>
until <logical expression>
```

While statement: repeating loop with the logical test happening at the beginning.

```
while <logical expression> do
  <statement or compound statement>
```

GoTo statement: (considered bad in structured programming by some computer scientists?)

```
goto <label>
```

Try and Raise statements: used to catch and raise exceptions

```
try
  <statement(s)>
except
  <exception handling statements>
end

try
  <statements>
finally
  <statements>
end

raise <exception>
```

C++ Statements

The C++ language includes support for several statement types including assignment, express, compound, if, switch, for, while, do, jump, continue, break and return.

Assignment statement: used to assign one variable to another variable or expression. Note: be careful about using = and == as = is used for assignment and == is used as a conditional operator.

```
variable = <variable or expression>;
```

Expression statement: The compiler executes an expression statement by evaluating the expression. All side effects from this evaluation are completed before the next statement is executed. Most expression statements are assignment statements or function calls. The null statement is a special case, consisting of a single semicolon (;). The null statement does nothing, and is therefore useful in situations where C++ syntax expects a statement but your program does not need one.

Labeled statement: used with the C++ goto statement. A labeled statement contains an identifier followed by a colon and a statement.

```
identifier : statement;
```

Compound statement or block: A compound statement, or block, is a list (possibly empty) of statements enclosed in matching braces ({ }). Syntactically, a block can be considered to be a single statement, but it also plays a role in the scoping of identifiers. An identifier declared within a block has a scope starting at the point of declaration and ending at the closing brace. Blocks can be nested to any depth up to the limits of memory.

If statement: Use if to implement a conditional statement.

```
if ( <condition1> )  <statement1>

if ( <condition1> )  <statement1>;
 else  <statement2>;

if ( <condition1> )  <statement1>;
 else if ( <condition2> )  <statement2>;
 else  <statement3>;

if ( <condition1> )
{
  if ( <condition2> ) {
  <statement1>
  <statement2>
  }
  else <statement3>
}
else
  <statement4>
```

Switch statement: Use the switch statement to pass control to a case that matches the <switch variable>. At which point the statements following the matching case evaluate. If no case satisfies the condition the default case evaluates. To avoid evaluating any other cases and relinquish control from the switch, terminate each case with **break**

```
switch ( <switch variable> ) {casebreakdefault
case <constant expression> : <statement>; [break;]
    .
    .
    .
default : <statement>;
}
```

For statement: The for statement implements an iterative loop.

```
for ( [<initialization>] ; [<condition>] ; [<increment>] )
<statement>
```

<condition> is checked before the first entry into the block.
<statement> is executed repeatedly UNTIL the value of <condition> is false.

Before the first iteration of the loop, <initialization> initializes variables for the loop. After each iteration of the loop, <increments> increments a loop counter. In C++, <initialization> can be an expression or a declaration. The scope of any identifier, declared within the for loop, extends to the end of the control statement only. A variable defined in the for-initialization expression is in scope only within the for-block. All the expressions are optional. If <condition> is left out, it is assumed to be always true.

While statement: Use the while keyword to conditionally iterate a statement.

```
while ( <condition> ) <statement>;
```

<statement> executes repeatedly until the value of <condition> is false. The test takes place before <statement> executes. Thus, if <condition> evaluates to false on the first pass, the loop does not execute.

Do statement: The do statement executes until the condition becomes false.

```
do <statement> while ( <condition> );
```

<statement> is executed repeatedly as long as the value of <condition> remains true.
Since the condition tests after each the loop executes the <statement>, the loop will execute at least once.

Jump Statements: A jump statement, when executed, transfers control unconditionally. There are four such statements: break, continue, goto, and return.

Break statement: Use the break statement within loops to pass control to the first statement following the innermost switch, for, while, or do block.

```
break;
```

Continue statement: Use the continue statement within loops to pass control to the end of the innermost enclosing end brace belonging to a looping construct, such as for or while; at which point the loop continuation condition is re-evaluated.

```
continue;
```

Goto statement: transfer control to a labeled statement.

```
goto identifier;
```

Return statement: the return statement exits the current function and returns control to the calling program passing back the value of an expression. If the function has a return type of **void** then the return statement can be used without a return value.

```
return <expression>;
```

Try block: Exception handling requires the use of three keywords: **try**, **throw**, and **catch**. A block of code in which an exception can occur must be prefixed by the keyword **try**. Following the **try** keyword is a block of code enclosed by braces. The **throw** keyword is used to generate an exception. The **try** block contains statements that might throw exceptions and is followed by one or more **catch** statements. The throw statement can throw various types of objects. Objects in C++ may generally be thrown by value, reference, or pointer. In addition, the **throw** statement can throw primitive types, such as integers or pointers. The **catch** statement has several forms. Objects may be caught by value, reference, or pointer. In addition, const modifiers can be applied to the catch parameter. There can be multiple catch statements for a single try block to allow a block to catch multiple different kinds of exceptions, and there should be a catch statement for each exception that might be thrown. In some cases, an exception handler may process an exception, then either rethrow the same exception or throw a different exception. If the handler wants to re-throw the current exception, it can just use the **throw** statement with no parameters. This instructs the compiler/RTL to take the current exception object and throw it again. The **__finally** keyword specifies actions that should be taken regardless of how the flow within the preceding try exits.

```
try
{
<statements. Can include throw exception>
}
catch (exception-declaration)
{
compound-statement
}
…
catch (exception-declaration)
{
compound-statement
}
__finally
{
```

```
compound-statement
}
```

This indicates that the program is prepared to test for the existence of exceptions. If an exception occurs, the program flow is interrupted. The sequence of steps taken is as follows:

- The program searches for a matching handler
- If a handler is found, the stack is unwound to that point
- Program control is transferred to the handler
- If no handler is found, the program will call the terminate function. If no exceptions are thrown, the program executes in the normal fashion.

The try, catch, and throw keywords are not allowed in C programs (use __try). For complete coverage of C++ exception handling syntax, read the try block information on the Embarcadero DocWiki at http://docwiki.embarcadero.com/RADStudio/en/Standard_C%2B%2B_Exception_Handling_Syntax

Functions (Delphi/C++) and Procedures (Delphi)

Both languages support building reusable routines: functions in Delphi and C++ and procedures in Delphi.

Delphi Procedures and Functions

Procedures and functions, referred to collectively as *routines*, are self-contained statement blocks that can be called from different locations in a program. A function is a routine that returns a value when it executes. A procedure is a routine that does not return a value. Function calls, because they return a value, can be used as expressions in assignments and operations.

When you declare a procedure or function, you specify its name, the number and type of parameters it takes, and, in the case of a function, the type of its return value; this part of the declaration is sometimes called the prototype, heading, or header. Then you write a block of code that executes whenever the procedure or function is called; this part is sometimes called the routine's body or block. You can read more about Delphi procedures and functions on the Embarcadero DocWiki at http://docwiki.embarcadero.com/RADStudio/en/Procedures_and_Functions

Procedure Declarations

A Delphi procedure declaration has the form:

```
procedure procedureName(parameterList); directives;
   localDeclarations;
begin
   statements
end;
```

where procedureName is any valid identifier, statements is a sequence of statements that execute when the procedure is called, and (parameterList), directives;, and localDeclarations; are optional.

Within a procedure's statement block, you can use variables and other identifiers declared in the localDeclarations part of the procedure. You can also use the parameter names from the parameter list; the parameter list defines a set of local variables, so don't try to redeclare the parameter names in the localDeclarations section. Finally, you can use any identifiers within whose scope the procedure declaration falls.

Function Declarations

A function declaration is like a procedure declaration except that it specifies a return type and a return value. Function declarations have the form:

```
function functionName(parameterList):returnType;directives;
   local Declarations;
begin
   statements
end;
```

where functionName is any valid identifier, returnType is a type identifier, statements is a sequence of statements that execute when the function is called, and the (parameterList), directives and local Declarations are optional.

The function's statement block is governed by the same rules that apply to procedures. Within the statement block, you can use variables and other identifiers declared in the localDeclarations part of the function, parameter names from the parameter list, and any identifiers within whose scope the function declaration falls. In addition, the function name itself acts as a special variable that holds the function's return value, as does the predefined variable Result.

You can assign a value to Result or to the function name repeatedly within a statement block, as long as you assign only values that match the declared return type. When execution of the function terminates, whatever value was last assigned to Result or to the function name becomes the function's return value. If the function exits without assigning a value to Result or the function name, then the function's return value is undefined.

C++ Functions

In the Kernighan and Ritchie style of declaration, a function could be implicitly declared by its appearance in a function call, or explicitly declared as follows

```
<type> func();
```

where type is the optional return type defaulting to int. In C++, this declaration means

```
<type> func(void);
```

A function can be declared to return any type except an array or function type. This approach does not allow the compiler to check that the type or number of arguments used in a function call match the declaration.

This problem was eased by the introduction of function prototypes with the following declaration syntax:

```
<type> func(parameter-declarator-list);
```

Note: You can enable a warning within the IDE or with the command-line compiler: "Function called without a prototype."

Declarators specify the type of each function parameter. The compiler uses this information to check function calls for validity. The compiler is also able to coerce arguments to the proper type. Suppose you have the following code fragment:

```
extern long lmax(long v1, long v2); /* prototype */
foo()
{
   int limit = 32;
   char ch = 'A';
   long mval;
   mval = lmax(limit,ch);    /* function call */
}
```

Since it has the function prototype for lmax, this program converts limit and ch to long, using the standard rules of assignment, before it places them on the stack for the call to lmax. Without the function prototype, limit and ch would have been placed on the stack as an integer and a character, respectively; in that case, the stack passed to lmax would not match in size or content what lmax was expecting, leading to problems. The classic declaration style does not allow any checking of parameter type or number, so using function prototypes aids greatly in tracking down programming errors.

Function prototypes also aid in documenting code. For example, the function strcpy takes two parameters: a source string and a destination string. The question is, which is which? The function prototype

```
char *strcpy(char *dest, const char *source);
```

makes it clear. If a header file contains function prototypes, then you can print that file to get most of the information you need for writing programs that call those functions. If you include an identifier in a prototype parameter, it is used only for any later error messages involving that parameter; it has no other effect.

A function declarator with parentheses containing the single word void indicates a function that takes no arguments at all:

```
func(void);
```

In C++, func() also declares a function taking no arguments.

A function prototype normally declares a function as accepting a fixed number of parameters. For functions that accept a variable number of parameters (such as printf), a function prototype can end with an ellipsis (...), like this:

```
f(int *count, long total, ...);
```

With this form of prototype, the fixed parameters are checked at compile time, and the variable parameters are passed with no type checking.

A function is called with actual arguments placed in the same sequence as their matching formal parameters. The actual arguments are converted as if by initialization to the declared types of the formal parameters.

Here is a summary of the rules governing how the compiler deals with language modifiers and formal parameters in function calls, both with and without prototypes:

- The language modifiers for a function definition must match the modifiers used in the declaration of the function at all calls to the function.
- A function can modify the values of its formal parameters, but this has no effect on the actual arguments in the calling routine, except for reference arguments in C++.

When a function prototype has not been previously declared, the compiler converts integral arguments to a function call according to the integral widening (expansion) rules described in Standard arithmetic conversions. When a function prototype is in scope, the compiler converts the given argument to the type of the declared parameter as if by assignment.

When a function prototype includes an ellipsis (...), the compiler converts all given function arguments as in any other prototype (up to the ellipsis). The compiler widens any arguments given beyond the fixed parameters, according to the normal rules for function arguments without prototypes.

If a prototype is present, the number of arguments must match (unless an ellipsis is present in the prototype). The types need to be compatible only to the extent that an assignment can legally convert them. You can always use an explicit cast to convert an argument to a type that is acceptable to a function prototype.
Note: If your function prototype does not match the actual function definition, the compiler will detect this if and only if that definition is in the same compilation unit as the prototype. If you create a library of routines with a corresponding header file of prototypes, consider including that header file when you compile the library, so that any discrepancies between the prototypes and the actual definitions will be caught. C++ provides type-safe linkage, so differences between expected and actual parameters will be caught by the linker.

Operators

Both languages include operators for binary arithmetic, logical operations, relational operations and pointer operations. Delphi also includes set and string operators.

Delphi Operators

Binary Arithmetic Operators

Operator	Operation	Operand Types	Result Type	Example
+	addition	integer, real	integer, real	X + Y
-	subtraction	integer, real	integer, real	Result -1
*	multiplication	integer, real	integer, real	P * InterestRate
/	real division	integer, real	real	X / 2
Div	integer division	integer	integer	Total div UnitSize
Mod	remainder	integer	integer	Y mod 6

Unary arithmetic operators

Operator	Operation	Operand Type	Result Type	Example
+	sign identity	integer, real	integer, real	+7
-	sign negation	integer, real	integer, real	-X

The following rules apply to arithmetic operators:

- The value of x / y is of type Extended, regardless of the types of x and y. For other arithmetic operators, the result is of type Extended whenever at least one operand is a real; otherwise, the result is of type Int64 when at least one operand is of type Int64; otherwise, the result is of type Integer. If an operand's type is a subrange of an integer type, it is treated as if it were of the integer type.
- The value of x div y is the value of x / y rounded in the direction of zero to the nearest integer.
- The mod operator returns the remainder obtained by dividing its operands. In other words,
- x mod y = x - (x div y) * y.
- A runtime error occurs when y is zero in an expression of the form x / y, x div y, or x mod y.

Boolean Operators

The Boolean operators not, and, or, and xor take operands of any Boolean type and return a value of type Boolean.

Operator	Operation	Operand Types	Result Type	Example
not	negation	Boolean	Boolean	not (C in MySet)

and	conjunction	Boolean	Boolean	Done and (Total >0)
or	disjunction	Boolean	Boolean	A or B
xor	exclusive disjunction	Boolean	Boolean	A xor B

Logical (Bitwise) Operators

The following logical operators perform bitwise manipulation on integer operands. For example, if the value stored in X (in binary) is 001101 and the value stored in Y is 100001, the statement:
Z := X or Y; assigns the value 101101 to Z.

Operator	Operation	Operand Types	Result Type	Example
not	bitwise negation	integer	integer	not X
and	bitwise and	integer	integer	X and Y
or	bitwise or	integer	integer	X or Y
xor	bitwise xor	integer	integer	X xor Y
shl	bitwise shift left	integer	integer	X shl 2
shr	bitwise shift right	integer	integer	Y shr I

The following rules apply to bitwise operators:

- The result of a not operation is of the same type as the operand.
- If the operands of an and, or, or xor operation are both integers, the result is of the predefined integer type with the smallest range that includes all possible values of both types.
- The operations x shl y and x shr y shift the value of x to the left or right by y bits, which (if x is an unsigned integer) is equivalent to multiplying or dividing x by 2^y; the result is of the same type as x. For example, if N stores the value 01101 (decimal 13), then N sh 1 returns 11010 (decimal 26). Note that the value of y is interpreted modulo the size of the type of x. Thus for example, if x is an integer, x shl 40 is interpreted as x shl 8 because an integer is 32 bits and 40 mod 32 is 8.

String operators

Operator	Operation	Operand Types	Result Type	Example
+	concatenation	string, packed string, character	string	S + '.'

The following rules apply to string concatenation:

- The operands for + can be strings, packed strings (packed arrays of type Char), or characters. However, if one operand is of type WideChar, the other operand must be a long string (UnicodeString, AnsiString or WideString).
- The result of a + operation is compatible with any string type. However, if the operands are both short strings or characters, and their combined length is greater than 255, the result is truncated to the first 255 characters.

Pointer Operators

The relational operators <, >, <=, and >= can take operands of type PAnsiChar and PWideChar (see Relational operators). The following operators also take pointers as operands. For more information about pointers, see Pointers and Pointer Types in Data Types, Variables, and Constants.

Operator	Operation	Operand Types	Result Type	Example
+	pointer addition	character ptr, integer	character ptr	P + I
-	pointer subtraction	character ptr, integer	character ptr, integer	P - Q
^	pointer dereference	pointer	base type of pointer	P^
=	equality	pointer	Boolean	P = Q
<>	inequality	pointer	Boolean	P <> Q

The ^ operator dereferences a pointer. Its operand can be a pointer of any type except the generic Pointer, which must be typecast before dereferencing.

P = Q is True just in case P and Q point to the same address; otherwise, P <> Q is True.

You can use the + and - operators to increment and decrement the offset of a character pointer. You can also use - to calculate the difference between the offsets of two character pointers. The following rules apply:

- If I is an integer and P is a character pointer, then P + I adds I to the address given by P; that is, it returns a pointer to the address I characters after P. (The expression I + P is equivalent to P + I.) P - I subtracts I from the address given by P; that is, it returns a pointer to the address I characters before P. This is true for PAnsiChar pointers; for PWideChar pointers P + I adds SizeOf(WideChar) to P.
- If P and Q are both character pointers, then P - Q computes the difference between the address given by P (the higher address) and the address given by Q (the lower address); that is, it returns an integer denoting the number of characters between P and Q. P + Q is not defined.

Set Operators

The following operators take sets as operands.

Operator	Operation	Operand Types	Result Type	Example
+	union	set	set	Set1 + Set2
-	difference	set	set	S - T
*	intersection	set	set	S * T
<=	subset	set	Boolean	Q <= MySet
>=	superset	set	Boolean	S1 >= S2
=	equality	set	Boolean	S2 = MySet
<>	inequality	set	Boolean	MySet <> S1
in	membership	ordinal, set	Boolean	A in Set1

The following rules apply to +, -, and *:

- An ordinal O is in X + Y if and only if O is in X or Y (or both). O is in X - Y if and only if O is in X but not in Y. O is in X * Y if and only if O is in both X and Y.
- The result of a +, -, or * operation is of the type set of A..B, where A is the smallest ordinal value in the result set and B is the largest.
- The following rules apply to <=, >=, =, <>, and in:
- X <= Y is True just in case every member of X is a member of Y; Z >= W is equivalent to W <= Z. U = V is True just in case U and V contain exactly the same members; otherwise, U <> V is True.
- For an ordinal O and a set S, O in S is True just in case O is a member of S.

Relational Operators

Relational operators are used to compare two operands. The operators =, <>, <=, and >= also apply to sets.

Operator/Operation/Operand Types/Result Type/Example

- = equality simple, class, class reference, interface, string, packed string Boolean I = Max
- <> inequality simple, class, class reference, interface, string, packed string Boolean X <> Y
- < less-than simple, string, packed string, PChar Boolean X < Y
- > greater-than simple, string, packed string, PChar Boolean Len > 0
- <= less-than-or-equal-to simple, string, packed string, PChar Boolean Cnt <= I
- >= greater-than-or-equal-to simple, string, packed string, PChar Boolean I >= 1

For most simple types, comparison is straightforward. For example, I = J is True just in case I and J have the same value, and I <> J is True otherwise. The following rules apply to relational operators:

- Operands must be of compatible types, except that a real and an integer can be compared.
- Strings are compared according to the ordinal values that make up the characters that make up the string. Character types are treated as strings of length 1.
- Two packed strings must have the same number of components to be compared. When a packed string with n components is compared to a string, the packed string is treated as a string of length n.
- Use the operators <, >, <=, and >= to compare PAnsiChar (and PWideChar) operands only if the two pointers point within the same character array.
- The operators = and <> can take operands of class and class-reference types. With operands of a class type, = and <> are evaluated according the rules that apply to pointers: C = D is True just in case C and D point to the same instance object, and C <> D is True otherwise. With operands of a class-reference type, C = D is True just in case C and D denote the same class, and C <> D is True otherwise. This does not compare the data stored in the classes. For more information about classes, see Classes and Objects.

The @ Operator

The @ operator returns the address of a variable, or of a function, procedure, or method; that is, @ constructs a pointer to its operand. For more information about pointers, see "Pointers and Pointer Types" in Data Types, Variables, and Constants. The following rules apply to @:

- If X is a variable, @X returns the address of X. (Special rules apply when X is a procedural variable; see "Procedural Types in Statements and Expressions" in Data Types, Variables, and Constants.) The type of @X is Pointer if the default {$T} compiler directive is in effect. In the {$T+} state, @X is of type ^T, where T is the type of X (this distinction is important for assignment compatibility, see Assignment-compatibility).
- If F is a routine (a function or procedure), @F returns F's entry point. The type of @F is always Pointer.
- When @ is applied to a method defined in a class, the method identifier must be qualified with the class name. For example,
- @TMyClass.DoSomething
- points to the DoSomething method of TMyClass. For more information about classes and methods, see Classes and Objects.
- Note: When using the @ operator, it is not possible to take the address of an interface method, because the address is not known at compile time and cannot be extracted at runtime.

Operator Precedence

In complex expressions, rules of precedence determine the order in which operations are performed.

```
Operators                      Precedence
@, not                         first (highest)
*,/,div,mod,and,shl,shr,as     second
+,-,or,xor                     third
=,<>,<,>,<=,>=,in,is           fourth (lowest)
```

C++ Operators

Operators are tokens that trigger some computation when applied to variables and other objects in an expression.

Assignment Operators

The assignment operators are:

```
*=    /=    %=
+=    -=    <<=   >>=
&=    ^=    |=
```

```
unary-expression  assignment-op  assignment-expression
```

The = operator is the only simple assignment operator, the others are compound assignment operators.

```
*=   Assign product
/=   Assign quotient
%=   Assign remainder (modulus)
+=   Assign sum
-=   Assign difference
<<=  Assign left shift
>>=  Assign right shift
&=   Assign bitwise AND
^=   Assign bitwise XOR
|=   Assign bitwise OR
```

Arithmetic operators

Use the C++ arithmetic operators to perform mathematical computations.

```
+ cast-expression
- cast-expression
add-expression + multiplicative-expression
add-expression - multiplicative-expression
multiplicative-expression * cast-expression
multiplicative-expression / cast-expression
multiplicative-expression % cast-expression
postfix-expression ++      (postincrement)
++ unary-expression        (preincrement)
postfix-expression --      (postdecrement)
-- unary-expression        (predecrement)
```

The unary expressions of + and - assign a positive or negative value to the cast-expression.

+ (addition), - (subtraction), * (multiplication), and / (division) perform their basic algebraic arithmetic on all data types, integer and floating point.

% (modulus operator) returns the remainder of integer division and cannot be used with floating points.

++ (increment) adds one to the value of the expression. Postincrement adds one to the value of the expression after it evaluates; while preincrement adds one before it evaluates.

-- (decrement) subtracts one from the value of the expression. Postdecrement subtracts one from the value of the expression after it evaluates; while predecrement subtracts one before it evaluates.

Bitwise Operators

Use the bitwise operators to modify the individual bits rather than the number.

```
AND-expression &  equality-expression
exclusive-OR-expression ^ AND-expression
inclusive-OR-expression exclusive-OR-expression
~ cast-expression
shift-expression << additive-expression
shift-expression >> additive-expression
```

- & bitwise AND; compares two bits and generates the result 1 if both bits are 1, otherwise it returns 0.
- | bitwise inclusive OR; compares two bits and generates the result 1 if either or both bits are 1, otherwise it returns 0.
- ^ bitwise exclusive OR; compares two bits and generates the result 1 if the bits are complementary, otherwise it returns 0.
- ~ bitwise complement; inverts each bit. ~ is used to create destructors.
- >> bitwise shift right; moves the bits to the right, discards the far right bit and if unsigned assigns 0 to the left most bit, otherwise sign extends.
- << bitwise shift left; moves the bits to the left, it discards the far left bit and assigns 0 to the right most bit.

Both operands in a bitwise expression must be of an integral type.

A	B	A & B	A ^ B	A | B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

Note: &, >>, << are context sensitive. & can also be the pointer reference operator.

Logical operators

Operands in a logical expression must be of scalar type.

```
logical-AND-expression && inclusive-OR-expression
logical-OR-expression || logical-AND-expression
! cast-expression
```

&& logical AND; returns true only if both expressions evaluate to be nonzero, otherwise returns false. If the first expression evaluates to false, the second expression is not evaluated.

|| logical OR; returns true if either of the expressions evaluate to be nonzero, otherwise returns false. If the first expression evaluates to true, the second expression is not evaluated.

! logical negation; returns true if the entire expression evaluates to be nonzero, otherwise returns false. The expression !E is equivalent to (0 == E).

Comma Operator

The comma separates elements in a function argument list.

```
expression , assignment-expression
```

The comma is also used as an operator in comma expressions. Mixing the two uses of comma is legal, but you must use parentheses to distinguish them.

The left operand E1 is evaluated as a void expression, then E2 is evaluated to give the result and type of the comma expression. By recursion, the expression

```
E1, E2, ..., En
```

results in the left-to-right evaluation of each Ei, with the value and type of En giving the result of the whole expression.

To avoid ambiguity with the commas in function argument and initializer lists, use parentheses. For example,

```
func(i, (j = 1, j + 4), k);
```

calls func with three arguments (i, 5, k), not four.

Relational Operators

Use relational operators to test equality or inequality of expressions. If the statement evaluates to be true it returns a nonzero character; otherwise it returns false (0).

```
relational-expression  <   shift-expression
relational-expression  >   shift-expression
relational-expression  <=  shift-expression
relational-expression  >=  shift-expression

Relational Operator Description
     >              greater than
     <              less than
     >=             greater than or equal to
     <=             less than or equal to
```

In the expression:

```
E1 <operator> E2
```

the operands must follow one of these conditions:

- Both E1 and E2 are of arithmetic type.
- Both E1 and E2 are pointers to qualified or unqualified versions of compatible types.

- Either E1 or E2 is a pointer to an object or incomplete type, and the other is a pointer to a qualified or unqualified version of void.
- Either E1 or E2 is a pointer, and the other is a null pointer constant.

Equality Operators

There are two equality operators in C++: == and !=. They test for equality and inequality between arithmetic or pointer values, following rules very similar to those for the relational operators.

```
equality-expression == relational-expression
equality-expression != relational-expression
```

Note: Notice that == and != have a lower precedence than the relational operators < and >, <=, and >=.

Also, == and != can compare certain pointer types for equality and inequality where the relational operators would not be allowed.

Conditional Operators

The conditional operator ? : is a ternary operator.

```
logical-OR-expression ? expression : conditional-expression
```

In the expression E1 ? E2 : E3, E1 evaluates first. If its value is true, then E2 evaluates and E3 is ignored. If E1 evaluates to false, then E3 evaluates and E2 is ignored.

The result of E1 ? E2 : E3 will be the value of either E2 or E3 depending upon which E1 evaluates.

E1 must be a scalar expression. E2 and E3 must obey one of the following rules:

1. Both of arithmetic type. E2 and E3 are subject to the usual arithmetic conversions, which determines the resulting type.
2. Both of compatible struct or union types. The resulting type is the structure or union type of E2 and E3.
3. Both of void type. The resulting type is void.
4. Both of type pointer to qualified or unqualified versions of compatible types. The resulting type is a pointer to a type qualified with all the type qualifiers of the types pointed to by both operands.
5. One operand is a pointer, and the other is a null pointer constant. The resulting type is a pointer to a type qualified with all the type qualifiers of the types pointed to by both operands.
6. One operand is a pointer to an object or incomplete type, and the other is a pointer to a qualified or unqualified version of void. The resulting type is that of the non-pointer-to-void operand.

Unary Operators

Unary operators group right-to-left.

```
<unary-operator> <unary expression>
<unary-operator> <type><unary expression>
```

The C++ language provides the following unary operators:

- ! Logical negation
- Indirection
- ~ Bitwise complement
- ++ Increment
- -- Decrement
- - Unary minus
- & Address-of (pointer dereference)

Plus and Minus Operators

Unary:

In these unary + - expressions

```
+ cast-expression
- cast-expression
```

the cast-expression operand must be of arithmetic type.

```
+ cast-expression - Value of the operand after any required
integral promotions.
- cast-expression: negative of the value of the operand
after any required integral promotions.
```

Binary:

- add-expression + multiplicative-expression
- add-expression - multiplicative-expression

Legal operand types for op1 + op2:

1. Both op1 and op2 are of arithmetic type.
2. op1 is of integral type, and op2 is of pointer to object type.
3. op2 is of integral type, and op1 is of pointer to object type.

In case 1, the operands are subjected to the standard arithmetical conversions, and the result is the arithmetical sum of the operands. In cases 2 and 3, the rules of pointer arithmetic apply.

Legal operand types for op1 - op2:

1. Both op1 and op2 are of arithmetic type.
2. Both op1 and op2 are pointers to compatible object types.
3. op1 is of pointer to object type, and op2 is integral type.

In case 1, the operands are subjected to the standard arithmetic conversions, and the result is the arithmetic difference of the operands. In cases 2 and 3, the rules of pointer arithmetic apply.

Note: The unqualified type <type> is considered to be compatible with the qualified types const type, volatile type, and const volatile type.

Reference/Dereference Operators

The & and * operators work together to reference and dereference pointers that are passed to functions.

```
& cast-expression
* cast-expression
```

Referencing operator (&) - Use the reference operator to pass the address of a pointer to a function outside of main().
The cast-expression operand must be one of the following:

- A function designator.
- An lvalue designating an object that is not a bit field and is not declared with a register storage class specifier.

If the operand is of type <type>, the result is of type pointer to <type>.

Some non-lvalue identifiers, such as function names and array names, are automatically converted into "pointer-to-X" types when they appear in certain contexts. The & operator can be used with such objects, but its use is redundant and therefore discouraged.

sizeof Operator

The sizeof operator has two distinct uses:

```
sizeof unary-expression
sizeof (type-name)
```

The result in both cases is an integer constant that gives the size in bytes of how much memory space is used by the operand (determined by its type, with some exceptions). The amount of space that is reserved for each type depends on the machine.

In the first use, the type of the operand expression is determined without evaluating the expression (and therefore without side effects). When the operand is of type char (signed or unsigned), sizeof gives the result 1. When the operand is a non-parameter of array type, the result is the total number of bytes in the array (in other words, an array name is not converted to a pointer type). The number of elements in an array equals sizeof array / sizeof array[0].

If the operand is a parameter declared as array type or function type, sizeof gives the size of the pointer. When applied to structures and unions, sizeof gives the total number of bytes, including any padding.

You cannot use sizeof with expressions of function type, incomplete types, parenthesized names of such types, or with an lvalue that designates a bit field object.

The integer type of the result of sizeof is size_t.

You can use sizeof in preprocessor directives; this is specific to C++Builder.

typeid Operator

You can use typeid to get run-time identification of types and expressions. A call to typeid returns a reference to an object of type const type_info. The returned object represents the type of the typeid operand.

```
typeid(expression)
typeid(type-name)
```

If the typeid operand is a dereferenced pointer or a reference to a polymorphic type, typeid returns the dynamic type of the actual object pointed or referred to. If the operand is non-polymorphic, typeid returns an object that represents the static type.

You can use the typeid operator with fundamental data types as well as user-defined types.

When the typeid operand is a class object/reference, typeid returns the static rather than run-time type.

If the typeid operand is a dereferenced NULL pointer, the bad_typeid exception is thrown.

Array Subscript Operator

Brackets ([ ]) indicate single and multidimensional array subscripts. The expression

```
<exp1>[exp2]
```

is defined as

```
*((exp1) + (exp2))
```

where either:

- exp1 is a pointer and exp2 is an integer or
- exp1 is an integer and exp2 is a pointer

Function Call Operator

Parentheses ( )

```
postfix-expression(<arg-expression-list>)
```

- group expressions
- isolate conditional expressions
- indicate function calls and function parameters

The value of the function call expression, if it has a value, is determined by the return statement in the function definition.

This is a call to the function given by the postfix expression.

arg-expression-list is a comma-delimited list of expressions of any type representing the actual (or real) function arguments.

. Direct Member Selection Operator

Use the selection operator (**.**) to access structure and union members.

-> Indirect Member Selection Operator

You use the selection operator -> to access structure and union members.

Increment / Decrement Operators

Increment operator (++)

```
postfix-expression ++        (postincrement)
++ unary-expression          (preincrement)
```

The expression is called the operand. It must be of scalar type (arithmetic or pointer types) and must be a modifiable lvalue..

Postincrement operator: The value of the whole expression is the value of the postfix expression before the increment is applied. After the postfix expression is evaluated, the operand is incremented by 1.

Preincrement operator: The operand is incremented by 1 before the expression is evaluated. The value of the whole expression is the incremented value of the operand. The increment value is appropriate to the type of the operand. Pointer types follow the rules for pointer arithmetic.

Decrement operator (--)

```
postfix-expression --       (postdecrement)
-- unary-expression         (predecrement)
```

The decrement operator follows the same rules as the increment operator, except that the operand is decremented by 1 after or before the whole expression is evaluated.

Primary Expression Operators

For ANSI C, a primary expression is literal (also sometimes referred to as constant), identifier, and (expression). The C++ language extends this list of primary expressions to include the keyword this, scope resolution operator ::, name, and the class destructor ~ (tilde).

The primary expressions are summarized in the following list.

```
primary-expression:
   literal:
   name:
   qualified-name: (C++ specific)
   qualified-class-name :: name
```

In nonstatic member functions, the keyword **this** is a pointer to the object for which the function is called. All calls to nonstatic member functions pass this as a hidden argument.

this is a local variable available in the body of any nonstatic member function. Use it implicitly within the function for member references. It does not need to be declared and it is rarely referred to explicitly in a function definition.. The keyword this cannot be used outside a class member function body.

The scope resolution operator :: allows reference to a type, object, function, or enumerator even though its identifier is hidden.

The parentheses surrounding an expression do not change the unadorned expression itself.

The primary expression name is restricted to the category of primary expressions that sometimes appear after the member access operators . (dot) and ->. Therefore, name must be either an lvalue or a function.

An identifier is a primary expression, provided it has been suitably declared. The description and formal definition of identifiers is shown in Identifiers.

Precedence of Operators

There are 16 precedence categories, some of which contain only one operator. Operators in the same category have equal precedence with each other.

Where duplicates of operators appear in the table, the first occurrence is unary, the second binary. Each category has an associativity rule: left to right, or right to left. In the absence of parentheses, these rules resolve the grouping of expressions with operators of equal precedence.

The precedence of each operator in the following table is indicated by its order in the table. The first category (on the first line) has the highest precedence. Operators on the same line have equal precedence.

```
Operators Associativity

()        left to right
[]
->
::
.
!         right to left
~
+
-
++
--
&
*
sizeof
new
delete
.*        left to right
->*
*         left to right
/
%
+         left to right
-
<<        left to right
>>
<         left to right
<=
>
>=
==        left to right
!=
&         left to right
^         left to right
|         left to right
&&        left to right
||        left to right
```

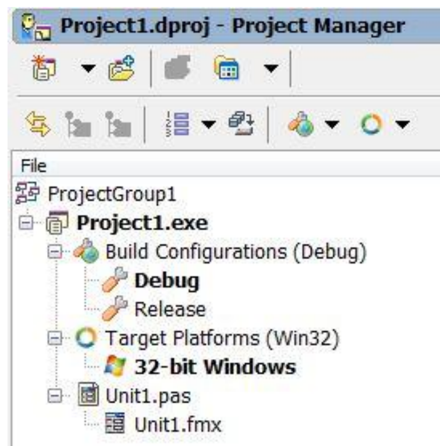
```
?:          right to left
=           right to left
*=
/=
%=
+=
-=
&=
^=
|=
<<=
>>=
,           left to right
```

Program Structure of a FireMonkey Application

You create a new Windows and Mac application with FireMonkey by using the File > New menu and select the "FireMonkey HD Application – Delphi", "FireMonkey 3D Application – Delphi", FireMonkey HD iOS Application – Delphi", "FireMonkey HD Application – C++Builder" or "FireMonkey 3D Application – C++Builder. After you make your choice, a starting Delphi or C++ project is created with a main program (.pas or .cpp extension), main form file (.fmx extension) and a source code unit (.pas for Delphi and .cpp for C++) and a header file (.h for C++ projects).

Example "FireMonkey HD Application – Delphi" project contents

Below you will find the contents of the Project Manager view after you have selected your starting project.



The Main Program source code file – Project1.pas

The main program program source file contains the following statements (Right-Mouse-Click on the Project1.exe node in the Project Manager and choose “View Source” from the context menu or hit Control-V when the Project1.exe node is selected) :

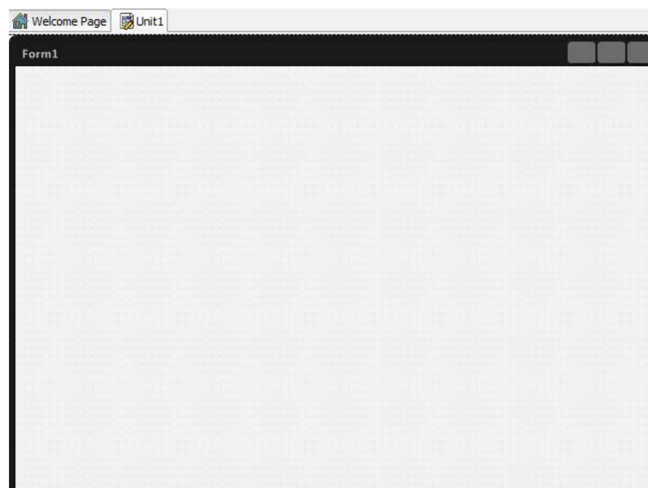
```
program Project1;

uses
  FMX.Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.res}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

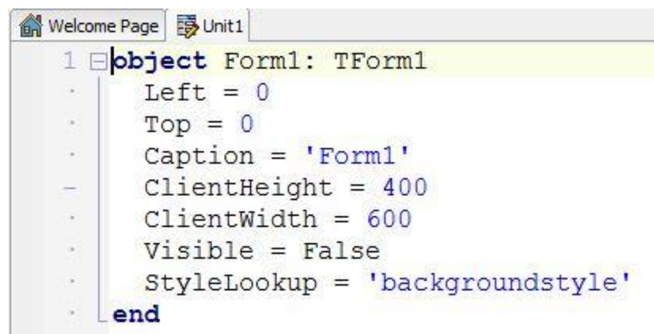
The “program” keyword denotes that your application is a program ☺. The “uses” clause contains a list of any separately compilable units that are needed to compile the program (more about units in a moment). The “{$R *.res}” compiler directive allows the Delphi compiler to include any resources (bitmaps, icons, cursors, etc) that are required for your application. The “begin” and “end.” Statements define the statements to be executed when the program starts. Notice that there will be at least one “Application.CreateForm” line for your main form in a FireMonkey application (there could be more if you have multiple forms as part of your application).

The Application.Run statement starts the application and leaves the rest of the execution and control to the form and any event handlers until you close (or quit) the application.

The Main Delphi Application Form – form view and text view

The main FireMonkey form for the application starts as an empty form with no components except the form itself. You can view it in the form designer and also in text mode in the code editor.





The Main Delphi Application Form – Unit Source code

Associated with the main application form is its unit1.pas source code file.

```
unit Unit1;

interface

uses
  System.SysUtils, System.Types, System.UITypes,
  System.Classes, System.Variants,
  FMX.Types, FMX.Controls, FMX.Forms, FMX.Dialogs;

type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;

implementation

{$R *.fmx}

end.
```

The unit keyword defines the start of the separately compilable (separate from the main program source code) source code file for the main form. Delphi units have two main sections: the Interface section and the Implementation section. The Interface section contains any declarations of types, constants and variables for the unit. The Implementation section contains the functions, procedures, methods and event handler application logic.

Unit1's Interface section contains a uses clause that includes a list of all of the FireMonkey and run-time library units that are required to compile the unit. The Interface section also includes the declaration of the TForm1 class which inherits from the TForm parent class. Delphi classes can have private, public and

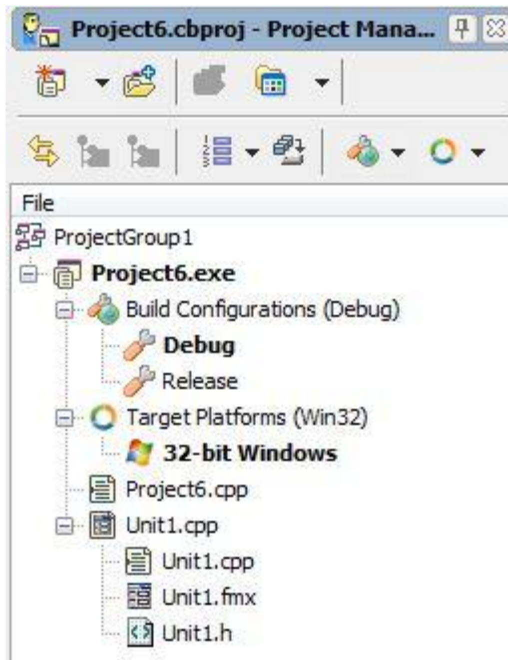
protected sections. As you add additional components using the form designer or the code editor, their declarations will be added to the TForm1 class.

The **var** section in the Interface section declares any global variables including the Form1 form variable. You can define any other global variables in this section as needed in your application logic.

The Implementation section when you first build the starting project only contains the {$R *.fmx} directive which tells the compiler to link the FireMonkey form file for this unit into the application executable. The "end." statement tells the compiler that this is the end of the unit.

As you add event handlers for the components you add to the form, you will see the event methods appear in the Interface section's TForm1 class declaration. A skeleton method implementation will also appear in the Implementation section.

Example "FireMonkey HD Application – C++Builder" project" contents



The Main C++ Project Source Code – Project6.cpp

The main program source contains the following statements (Right-Mouse-Click on the Project6.exe node in the Project Manager and choose "View Source" from the context menu or hit Control-V when the Project6.exe node is selected):

```
//---------------------------------------------------------------------------
#include <fmx.h>
#pragma hdrstop
#include <tchar.h>
```

```
//---------------------------------------------------------------------------
USEFORM("Unit1.cpp", Form1);
//---------------------------------------------------------------------------
extern "C" int FMXmain()
{
	try
	{
		Application->Initialize();
		Application->MainFormOnTaskBar = true;
		Application->CreateForm(__classid(TForm1),
		                          &Form1);
		Application->Run();
	}
	catch (Exception &exception)
	{
		Application->ShowException(&exception);
	}
	catch (...)
	{
		try
		{
			throw Exception("");
		}
		catch (Exception &exception)
		{
			Application->ShowException(&exception);
		}
	}
	return 0;
}
//---------------------------------------------------------------------------
```

In the main program source code you will find the following lines:

#include

The “#include <fmx.h>” tells the compiler to bring in FireMonkey header file declarations that are needed to be able to compile the code. In C and C++ header files usually contain the declaration of types, classes, constants and other items used in the implementation code.

```
#include <header_name>
#include "header_name"
#include <macro_definition>
```

The #include directive pulls in other named files (known as include files, header files or headers) into the source code. The syntax has three versions:

- The first and second versions imply that no macro expansion will be attempted; in other words, header_name is never scanned for macro identifiers. header_name must be a valid file name with an extension (traditionally .h for header files) and optional path name and path delimiters.

- The third version assumes that neither < nor " appears as the first non-whitespace character following #include; further, it assumes a macro definition exists that will expand the macro identifier into a valid delimited header name with either of the <header_name> or "header_name" formats.

The preprocessor removes the #include line and conceptually replaces it with the entire text of the include file at that point in the source code. The source code itself is not changed, but the compiler "sees" the enlarged text. The placement of the #include can therefore influence the scope and duration of any identifiers in the included file. If you place an explicit path in the header_name, only that directory will be searched.

The difference between the <header_name> and "header_name" formats lies in the searching algorithm employed in trying to locate the include file:

Quoted form - The "header_name" version specifies a user-supplied include file; the file is searched in the following order:

- The same directory of the file that contains the #include statement.
- The directories of files that include (#include) that file.
- The current directory.
- The path specified by the Include file search path (-I) option.

Angle-bracket form - The <header_name> version specifies a standard include file. The search is made successively in each of the include directories in the order they are defined by the Include file search path (-I) option. If the file is not located in any of the default directories, an error message is issued.

#pragma

With #pragma, you can set compiler directives in your source code, without interfering with other compilers that also support #pragma. If the compiler doesn't recognize directive-name, it ignores the #pragma directive without any error or warning message.

The “#pragma hdrstop” line tells the compiler to stop creating “Pre-Compiled Headers”. Creating and using precompiled headers can do two major things for you:

- Can reduce the compilation time of C++ files
- Can reduce the number of lines of code that the compiler must process (in some cases, by several orders of magnitude)

You can read more about Pre-Compiled Headers on the Embarcadero DocWiki at
http://docwiki.embarcadero.com/RADStudio/en/Precompiled_Headers_Overview

USEFORM("Unit1.cpp", Form1)

USEFORM is a C++ macro that lists a form name (“Form1”) and the source code unit associated with the form (“Unit1.cpp”).

extern "C" int FMXmain()

extern "C" is used to prevent a program's function names from being mangled in C++ programs. This line defines the name of the FireMonkey main program function with a return type of "int" or integer.

try, throw, catch

C++ exception handling requires the use of three keywords: try, throw, and catch. The throw keyword is used to generate an exception. The try block contains statements that might throw exceptions and is followed by one or more catch statements. Each catch statement handles a specific type of exception.

The try block contains a statement or statements that can throw an exception. A program throws an exception by executing a throw statement. The throw statement generally occurs within a function. A try block specified by try must be followed immediately by the handler specified by catch. The try block is a statement that specifies the flow of control as the program executes. If an exception is thrown in the try block, program control is transferred to the appropriate exception handler.

The handler is a block of code designed to handle the exception. The C++ language requires at least one handler immediately after a try block. The program should include a handler for each exception that the program can generate. The throw statement can throw various types of objects. Objects in C++ may generally be thrown by value, reference, or pointer. In addition, the throw statement can throw primitive types, such as integers or pointers.

The catch statement has several forms. Objects may be caught by value, reference, or pointer. In addition, const modifiers can be applied to the catch parameter. There can be multiple catch statements for a single try block to allow a block to catch multiple different kinds of exceptions, and there should be a catch statement for each exception that might be thrown.

With multiple catch statements for a single try statement, you can have handlers for each type of exception. If an exception object is derived from some base class, you may want to add specialized handlers for some derived exceptions, but also include a generic handler for the base class. You do this by placing the catch statements in the order that you want them to be searched when an exception is thrown. If you want your handler to catch all exceptions that might be thrown past the try block, use the catch(...) special form. This tells the exception handling system that the handler should be invoked for any exception.

Application->

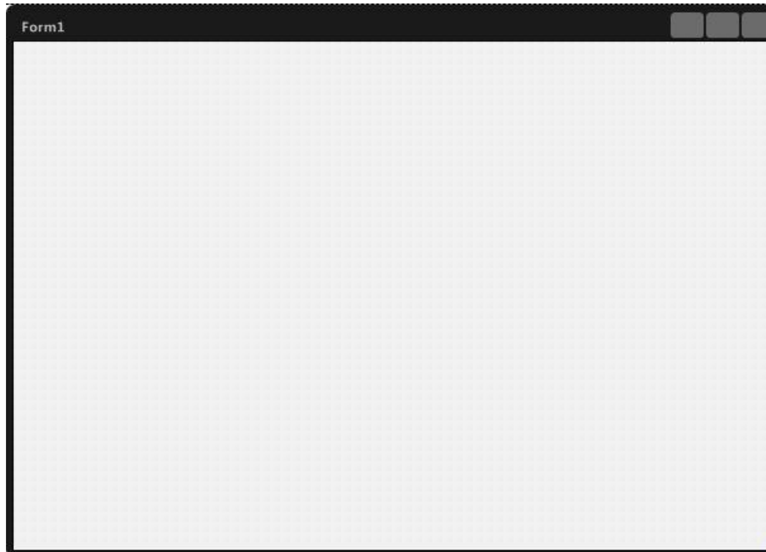
You will see several lines in the main program that use the Application object and call it's methods to initialize the application, create the main form (you can have multiple forms created in the main program or you can create forms on the fly in your code) and "run" the application. Your FireMonkey will continue to execute until you terminate the application with some user event (like a File |Exit menu item) or when a runtime exception occurs.

return 0

The return statement is used to exit from the main program (or current function) back to the calling routine (in this case, back to the operating system), optionally returning a value. The value zero denotes a successful termination of the program. You can return any other value to let the

The Main C++ Application Form – form view and text view

The main FireMonkey form for the application starts as an empty form with no components except the form itself. You can view it in the form designer and also in text mode in the code editor.



Welcome Page | Unit1 | Project6

```
object Form1: TForm1
  Left = 0
  Top = 0
  Caption = 'Form1'
  ClientHeight = 400
  ClientWidth = 600
  Visible = False
  StyleLookup = 'backgroundstyle'
end
```

In the form designer you can add additional visual and non-visual components to your application and they will appear in the form. You can use the Object Inspector to customize your form and the components it contains.

The Main C++ Application Form – Unit Source Code and Header File

Associated with the main application form is its unit1.cpp source code.

```
//---------------------------------------------------------------------------
#include <fmx.h>
#pragma hdrstop
#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.fmx"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
     : TForm(Owner)
{
}
//---------------------------------------------------------------------------
```

In C++ the code to be compiled is put in the source file. The declarations are placed in the header (.h) file.

```
//---------------------------------------------------------------------------
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------------
#include <System.Classes.hpp>
#include <FMX.Controls.hpp>
#include <FMX.Forms.hpp>
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
__published:   // IDE-managed Components
private:       // User declarations
public:        // User declarations
     __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

Using Forms

When you create a form in the IDE, Delphi and C++Builder automatically create the form in memory by including code in the main entry point of your application function. Usually, this is the desired behavior and you don't have to do anything to change it. That is, the main window persists through the duration of your program, so you would likely not change the default behavior when creating the form for your main window.

However, you may not want all your application's forms in memory for the duration of the program execution. That is, if you do not want all your application's dialogs in memory at once, you can create the dialogs dynamically when you want them to appear.

Forms can be modal or modeless. Modal forms are forms with which the user must interact before switching to another form (for example, a dialog box requiring user input). Modeless forms are windows that are displayed until they are either obscured by another window or until they are closed or minimized by the user.

You have full control in your application code to auto-create forms and create forms dynamically. The Embarcadero DocWiki has several articles about using forms, creating forms and controlling forms:

- Controlling When Forms Reside in Memory - http://docwiki.embarcadero.com/RADStudio/XE2/en/Controlling_When_Forms_Reside_in_Memory
- Displaying an Auto-created Form - http://docwiki.embarcadero.com/RADStudio/XE2/en/Displaying_an_Auto-created_Form
- Creating Forms Dynamically - http://docwiki.embarcadero.com/RADStudio/XE2/en/Creating_Forms_Dynamically
- Creating Modeless Forms - http://docwiki.embarcadero.com/RADStudio/XE2/en/Creating_Modeless_Forms_Such_as_Windows
- Creating a Form Instance Using a Local Variable - http://docwiki.embarcadero.com/RADStudio/XE2/en/Creating_a_Form_Instance_Using_a_Local_Variable
- Retrieving Data from Forms - http://docwiki.embarcadero.com/RADStudio/XE2/en/Retrieving_Data_from_Forms
- Retrieving Data from Modal Forms - http://docwiki.embarcadero.com/RADStudio/XE2/en/Retrieving_Data_from_Modal_Forms
- Retrieving Data from Modeless Forms - http://docwiki.embarcadero.com/RADStudio/XE2/en/Retrieving_Data_from_Modeless_Forms

FMX.Forms.TForm

TForm represents a standard FireMonkey application window (form). When you create forms in the Form designer at design time, they are implemented as descendants of TForm. Forms can represent the application's main window, or dialog boxes or various preferences-related windows. A form can contain any other FireMonkey objects, such as TButton, TCheckBox, TComboBox objects, and so on.

Exploring one of the Delphi and C++ example programs – ControlsDemo

A great way to learn about programming languages and using FireMonkey to build Windows and Mac applications is to read sample projects and the source code.

You will find the “Controls Demo” for Delphi and C++ at

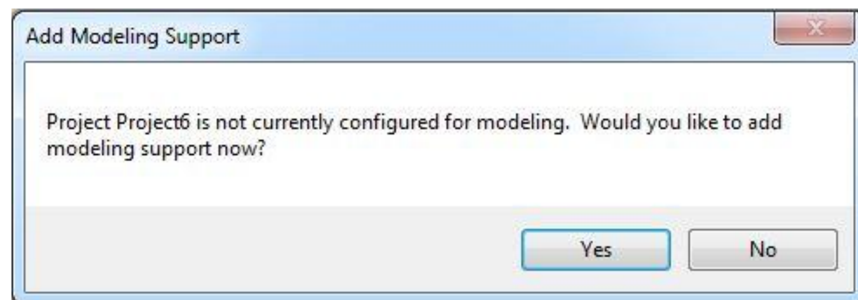
- C:\Users\Public\Documents\RAD Studio\9.0\Samples\FireMonkey\ControlsDemo
- C:\Users\Public\Documents\RAD Studio\9.0\Samples\CPP\FireMonkey\ControlsDemo

You can explore the source code yourself or watch the Lesson 4 replay video.

Object Design (using the UML Class Diagram)

Another great way to learn about programming languages and using FireMonkey to build Windows and Mac applications is to use the Model View and look at the definition of the classes used in a program. You can also use the Model View to create and modify classes.

The Model View shows the model tree of logical structure and containment hierarchy of your modeling project. To open the Model View, use the **View > Model View** menu command or click on the "Model View" tab at the bottom of the Project Manager Window. If this is the first time you have chosen the model view, you will see the following dialog box:



Click "Yes" and the IDE will process the project source and header files to create the model tree. None of your source code will be harmed. The modeling technology creates XML files containing the information about your classes. The model tree provides the logical representation of the model of your modeling project. The model tree shows the hierarchy of modeling elements in your projects: the root nodes are project nodes, then come nodes of namespaces (packages) and diagrams, then nodes of other modeling elements shown on diagrams. Double-click a node in the Model View tree to open the Code Editor for a specific class, interface, or member. Note: Double-clicking a namespace node in the Model View tree cannot open a specific source code file, since namespaces can span multiple source files.

The Model View provides the context menu that you can show by right-clicking model tree nodes. Commands available on the context menu depend on the selected modeling element. Using the context menu, you can add new elements to the model directly in the Model View. Some of the actions provided by the context menu are

- Opening UML diagrams in the Diagram View (Delphi, for C++ use the C++ Class Explorer)
- Adding (or deleting) new UML diagrams in a project (Delphi)
- Adding (or deleting) modeling elements on a UML diagram (Delphi)
- Copying, cutting, and pasting modeling elements
- Creating hyperlinks
- Adding constraints
- Going to declaration of the selected modeling element in the Code Editor
- Locating the modeling element, selected in the Model View, on the corresponding UML diagram in the Diagram View

- Adding user properties.
- Generating documentation
- And other operations

RAD Studio's modeling allows you to change your classes in the code editor and in the diagram view (since the model is just a view on the source code, both are kept in sync).

Handling Windows and Mac Platform Specific Code

There are several ways to handle code that needs to do something specific on Windows and something different on the Mac. Two common ways are to 1) use conditional compilation (compile time) and 2) use the TOSVersion record.

Conditional Compilation

Conditional compilation is based on the existence and evaluation of constants, the status of compiler switches, and the definition of conditional symbols. Conditional symbols work like Boolean variables: they are either defined (true) or undefined (false). Any valid conditional symbol is treated as false until it has been defined.

For Delphi, the {$DEFINE} directive sets a specified symbol to true, and the {$UNDEF} directive sets it to false. You can also define a conditional symbol by using the -D switch with the command-line compiler or by adding the symbol to the Conditional Defines field on the **Project > Options > Delphi Compiler** page. Delphi's conditional directives {$IFDEF}, {$IFNDEF}, {$IF}, {$ELSEIF}, {$ELSE}, {$ENDIF}, and {$IFEND} allow you to compile or suppress code based on the status of a conditional symbol. {$IF} and {$ELSEIF} allow you to base conditional compilation on declared Delphi identifiers. {$IFOPT} compiles or suppresses code depending on whether a specified compiler switch is enabled. The Delphi predefined Conditionals are listed on the Embarcadero DocWiki at http://docwiki.embarcadero.com/RADStudio/XE2/en/Conditional_compilation_(Delphi)

For C++, the compiler supports conditional compilation by replacing the appropriate source-code lines with a blank line. The lines thus ignored are those lines that are not to be compiled as a result of the directives. All conditional compilation directives must be completed in the source or include file in which they are begun. C++'s conditional directives are #if, #ifdef, #ifndef, #elif, #else, and #endif. You can use the #ifdef and #ifndef directives instead of the #if defined identifier expressions. However, the #if defined identifier expressions are preferred. The #ifdef and #ifndef directives are provided only for compatibility with previous versions of the language. The #ifdef and #ifndef conditional directives let you test whether an identifier is currently defined or not. That is, whether a previous #define command has been processed for that identifier and is still in force. You can find more information on the Embarcadero DocWiki at http://docwiki.embarcadero.com/RADStudio/en/If,_elif,_else,_And_endif

For Delphi and C++ FireMonkey applications, there are conditional define variables set for Windows and Mac. An example from the FireMonkey Delphi source code is

```
{$IFDEF MACOS}
```

```
    or ((Button = TMouseButton.mbLeft)
    and (Shift = [ssLeft, ssCtrl]))
{$ENDIF}
```

Syntax for C++ conditional compilation:

```
#if constant-expression-1
<section-1>
<#elif constant-expression-2 newline section-2>
  .
  .
  .
<#elif constant-expression-n newline section-n>
<#else <newline> final-section>
#endif
```

TOSVersion

RAD studio also defines a runtime structure called “TOSVersion” which is defined in the System.SysUtils unit of the runtime library. TOSVersion contains information about the operating system, OS version information and the processor. You can use TOSVersion in your program code to execute platform specific code at runtime:

- Platform (Windows or MAC OS X): pfWindows, pfMacOS
- Architecture (Intel x86 or Intel x64): arIntelX86, arIntelX64

You can find the Delphi and C++ definition of TOSVersion on the Embarcadero DocWiki at http://docwiki.embarcadero.com/Libraries/XE2/en/System.SysUtils.TOSVersion

TOSVersion has two methods you can use:

- Check - Returns whether the version of the current operating system is above or equal to a specific value.
- ToString - Returns the string representation of TOSVersion.

Summary, Looking Forward, To Do Items, Resources, Q&A and the Quiz

In Lesson 4, we took at look at what is inside the project and source code of a FireMonkey application. We discussed object oriented programming, classes, objects properties, methods, events and more. Along the way we also saw some of the basic language syntax and how to “read” source code in the editor and also in the model view. It is impossible to learn the Delphi or C++ programming language in 30 minutes. Thankfully, there are many tutorials, videos, books and articles to help you take advantage of each programming language.

In Lesson 5, you will create a Windows and Mac FireMonkey HD application, its user interface (UI) and the components you can use to build great looking applications.

In the meantime, here are some things to do, articles to read and videos to watch to enhance what you learned in Lesson 4 and to prepare you for lesson 5.

To Do Items

- Learn more about the Delphi and C++ languages using the additional resources links below.
- Create your own starting Delphi or C++ FireMonkey project. Take a look at the source code that is generated by the project wizard.
- Load and read through some of the example Delphi and C++ projects that are included with RAD Studio. You'll find them on your hard drive at C:\Users\Public\Documents\RAD Studio\9.0\Samples. We also have all of the latest versions of and new samples on Source Forge at http://radstudiodemos.svn.sourceforge.net/viewvc/radstudiodemos/branches/RadStudio_XE2/

Recommended Reading – Books and eBooks

The following books and e-books (commercial and free) are recommended for Delphi and C++ developers.

Delphi

Delphi XE2 Books and eBooks for sale by Bob Swart: These books offer hundreds of pages that will save you weeks of time and effort by learning from Bob's real-world experiences. Bob's books are available for purchase in PDF format and some are available in print form on Lulu.com - http://www.lulu.com/shop/search.ep?contributorId=300146).

- Delphi XE2 Development Essentials (145 pages), see http://www.drbob42.com/courseware/51.htm
- Delphi XE2 iOS Development (150 pages), see http://www.drbob42.com/courseware/56.htm
- Delphi XE2 DataSnap Development (276 pages), see http://www.drbob42.com/courseware/53.htm
- Delphi XE2 XML, SOAP and Web Services (152 pages), see http://www.drbob42.com/courseware/52.htm

Delphi Handbooks by Marco Cantu: Marco has been writing about Delphi for years. Many of Marco's books are available in print form on Amazon and in PDF format. Here is a list of Marco's Handbooks, Essential Delphi and Essential Pascal books and e-books.

- Delphi XE Handbook - http://www.marcocantu.com/handbooks/#dxeh
- Delphi 2010 Handbook - http://www.marcocantu.com/dh2010
- Delphi 2009 Handbook - http://www.marcocantu.com/dh2009/
- Delphi 2007 Handbook - http://www.marcocantu.com/dh2007/
- Delphi Handbooks Collection (Second Edition) e-Book PDF - http://sites.fastspring.com/wintechitalia/product/delphihandbookscollection
- Essential Pascal - http://www.marcocantu.com/epascal/default.htm

C++

- Thinking in C++ by Bruce Eckel (Second Edition), free eBooks Volume 1 (Bruce Eckel) and Volume 2 (Bruce Eckel and Chuck Allison) - http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html
- Thinking in C++ Solution Volume 1 Solution Guide ($12 USD, e-book only) - http://www.mindview.net/Books/TICPP/Solutions/

Links to Additional Resources

- Getting Started Course landing page - http://www.embarcadero.com/firemonkey/firemonkey-e-learning-series
- "Coming soon to a RAD IDE near you, the future of C++ - 64bit, C++11, ARM, iOS and Android" by: John Ray Thomas - http://edn.embarcadero.com/article/42275

Delphi:

- Embarcadero DocWiki - Delphi Reference - http://docwiki.embarcadero.com/RADStudio/en/Delphi_Reference
- Delphi Basics - http://www.delphibasics.co.uk/
- About Delphi Programming - http://delphi.about.com/
- A Beginner's Guide to Delphi - http://delphi.about.com/od/beginners/a/delphicourse.htm
- Introduction to Delphi - http://www.functionx.com/delphi/Lesson01.htm

C++:

- A Quick Introduction to C++ by Tom Anderson (29 pages) http://www.cs.washington.edu/homes/tom/c++example/c++.pdf
- MIT Open Courseware: Introduction to C++ - http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-096-introduction-to-c-january-iap-2011/
- C++ Language Tutorial - http://www.cplusplus.com/doc/tutorial/
- Bjarne Stroustrup – The C++ Programming Language home page - http://www2.research.att.com/~bs/C++.html

- Embarcadero DocWiki C++ Reference - http://docwiki.embarcadero.com/RADStudio/en/C%2B%2B_Reference
- C++Builder tutorials and VCL topics – a little dated, but an easy place to learn - http://www.functionx.com/bcb/index.htm
- C++ Builder Practical Learning Series – again a little dates but still classic tutorials - http://www.yevol.com/en/bcb/
- C++Builder Developer's Journal - http://bcbjournal.org/
- Programming Abstractions – Stanford, Instructor - Julie Zelenski - successor to Stanford iTuneU "Programming Methodology" course. Covers such advanced programming topics as recursion, algorithmic analysis, and data abstraction using the C++ programming language - http://itunes.apple.com/WebObjects/MZStore.woa/wa/viewPodcast?id=384232917

Q&A:

Here are the answers for the questions I've received for this lesson. I will continue to update this Course Book if I receive additional questions.

Q: Look forward to when FireMonkey based programs can be a DataSnap server.
A: You can build DataSnap servers in Delphi and C++ for Windows today. You can build FireMonkey client applications on Windows and Mac that talk to DataSnap servers running on Windows. Delphi developers can also build FireMonkey iOS client applications that can talk to Delphi and C++ DataSnap servers running on Windows. See Anders Ohlsson's blog at http://blogs.embarcadero.com/ao/category/ios for examples of iOS development with DataSnap.

Q: When will a new roadmap be published?
A: We presented information about next generation Delphi compilers for ARM processors when we released Delphi XE2 (the slide was titled "Beyond XE2). We also have EDN articles that talk about future compiler work. Recently, we also released an article about the C++ roadmap that includes 64-bit Windows support (something that Delphi already has) and iOS and Android compiler support. These same capabilities will be available for Delphi and C++ in the coming months and years.

Q: Can you please explain the tools you used in windows to emulate MAC?
A: Our IDE runs on Windows. We connect to our Platform Assistant (PAServer) running on Macintosh (via TCP/IP and a port) to be able to deploy and debug the application from the Windows IDE. We are not emulating the Mac environment. In my case, I am running MacOS X Lion, VMWare Fusion for the Mac which allows me to run Windows 7 in a guest virtual machine. Applications built for MacOS X are compiled as native code Mac applications and executed on the Mac operating system.

Q: In the future, will the FireMonkey be based on Windows Jupiter platform OR will it have its own libraries to generate HD?
A: FireMonkey uses DirectX to render HD and 3D applications on Windows. FireMonkey uses CoreGraphics Library for HD and OpenGL for 3D for Macintosh applications. FireMonkey for iOS uses CoreGraphics library for HD and OpenGL/ES for 3D.

Q: Will FireMonkey support Linux in the future?

A: Yes, Linux is on our FireMonkey roadmap. We are working on iOS and Android support with our next generation compilers first.

Q: Now that I see the {$ifdef FPC} compiler directives in the FMX files. I'm wondering if I can development my basic Fire Monkey apps in Windows and then recompile them under linux with FPC?
A: We do not have an implementation of FireMonkey for Linux HD and 3D graphics yet.

Q: Looking on Dave's screen, I don't see "Rave" in the Tool Palette. Is Rave not included in Delphi XE2, or just not available in a FireMonkey project? Note that Rave can produce PDF files, which per Lesson 1, is one means of doing cross platform reports. I got to be able to do reports.
A: Rave is part of the installation choices for Delphi XE2. Also included is FastReport for Delphi XE2. I guess I didn't not choose Rave as part of my installation choices. Rave is only available currently for Windows. Fast Reports has announced a FireMonkey version that will run on Windows and Mac.

Q: What is the difference between protected and public?
A: Public = everyone has access to these. Protected = descendants have access (and anything in the same unit).

Q: When do I use public and when would i want to use protected?
A: Use Public when you want everyone to have access. Private is for internals that no one else should have access to. Protected is for internals that you think that descendants may need access to, but you do not want to make public. Published is for properties and events that you want to surface in the Object Inspector.

Q: Can Delphi produce compressed executables?
A: We don't provide an executable compression tool. You can use any third party executable compression tools that might be available.

Q: Are we also going to do something with IntraWeb?
A: IntraWeb is created by Atozed Software. They are responsible for the continued development of IntraWeb. IntraWeb is still included in Delphi XE2, C++Builder XE2 and RAD studio XE2 for building application style web applications. We are very happy that Atozed Software continues to innovate in building components that allows the same desktop application style programming for web applications.

Q: Are there any educational resources for Diagrams? They seem to get very messy very quickly once there is a complex class hierachy. I am guessing there must be an elegant way of using them for real world applications?
A: In the UML class diagram view there is a layout choice to clean up the layout of classes. You can find a lot of information about UML modeling and the diagrams in the online help and the Embarcadero DocWiki. There is also a wealth of information about UML at the http://www.uml.org web site.

Q: How much overhead is added when using modeling?
A: There is no overhead added to your source code or applications. Using the model view creates XML files on disk with information about your source code. No source code is modified. No code is added to your application. The Model View is just a view of the source code.

Q: Since data-aware grids are so near and dear to business apps, are you going to cover the options in FireMonkey?

A: We are enhancing the FireMonkey platform to add additional interfaces to allow third party component vendors to create advanced data grids. At the same time, with FireMonkey's ability to create composite components you can add capabilities to FireMonkey grids. Check out Stephen Ball's recent blog post, "Getting to grips with using FireMonkey Grids" at http://blogs.embarcadero.com/stephenball/2012/05/29/getting-to-grips-with-using-firemonkey-grids/. Stephen says "I also love the ability to embed controls into the grid which is a lot easier than the OwnerDraw work we had to do before. This coupled with the different grid columns really does make for quite a useful grid indeed."

Q: What are some good client/server type databases that can be used for both Win and Mac?
A: InterBase server runs on both Windows, Mac, Linux and Solaris. InterBase XE Developer Edition ships with Delphi XE2, C++Builder XE2 and RAD Studio XE2. InterBase XE Developer Edition is free for application development work. InterBase Desktop, InterBase Server and InterBase ToGo (InterBase server in a DLL) editions are available for deployment. You'll find complete InterBase information at http://www.embarcadero.com/products/interbase.

Q: Do you plan update help for Indy in RAD? Help files very poor.
A: The Project Indy (Internet Direct) open source team develops Indy. We ship Indy in our products to give developers a large set of components for Internet application development. Project Indy components are available for Windows and Mac. You'll find the latest Indy information and demos at http://www.projectindy.org/index.en.aspx.

Q: Any plans for a PLE of XE2?
A: We have our Delphi XE2 and C++Builder XE2 starter editions are a great way to get started building high-performance applications for Windows. Delphi and C++Buidler Starter Editions includes a streamlined IDE, code editor, ultra fast compiler, integrated debugger, two-way visual designers to speed development, hundreds of visual components (FireMonkey and VCL), local connectivity with the InterBase database, and a limited commercial use license. You can find additional information at http://www.embarcadero.com/products/delphi/starter and http://www.embarcadero.com/products/cbuilder/starter.

Q: I couldn't find TActionManager as per the instructions. Is this not available in the trial, or has it been replaced or am I missing something?
A: Actions, Action Manager, Gesture/Touch, Video, Audio and other new capabilities are on the roadmap to be added to future releases of FireMonkey.

Q: Is there a view so that I can see the model and the code at the same time?
A: You can dock the model view in the IDE below the source code view. But you can't doc the class diagram away from the Code editor. You can't doc the Class Diagram separate from the code editor window without using another third party IDE tool. You can also use the Delphi Class Explorer and C++ Class Explorer in separate windows alongside the code editor.

Q: What is your roadmap support for Windows Phone 7?
A: You can use Embarcadero Prism XE2 to build applications for Windows Phone 7 and Windows 8 Metro. Embarcadero Prism is part of RAD Studio.

Q: If the app runs on the Mac, does it run on the iPad or iPod Touch or iPhone?

A:iOS devices use the ARM processor. Macs use the Intel processor. We have an implementation of FireMonkey for iOS using Delphi and we leverage the Free Pascal Compiler to compile the FireMonkey and Delphi code for iOS devices running ARM processors. Just because an application runs on Macintosh doesn't mean you can also run the same program on iOS. You need to compile the FireMonkey project for Mac and for iOS to create the native code applications for each Apple Platform. We use cross compiler technologies and tools to provide platform support for building Windows, Mac and iOS applications that you can deploy to each platform. In the future you will be able to cross compile all FireMonkey target application projects to Windows, Mac and iOS from your Windows-based IDE.

Q: Is it possible to access WMI-information in FireMonkey like Magenta Systems' WMI-package (http://www.magsys.co.uk/delphi/magwmi.asp)? Are there plans or supporting something like it for the Mac?
A: From WikiPedia: "Windows Management Instrumentation (WMI) - http://msdn.microsoft.com/en-us/library/windows/desktop/aa394582(v=vs.85).aspx - is a set of extensions to the Windows Driver Model. WMI is Microsoft's implementation of the Web-Based Enterprise Management (WBEM) and Common Information Model (CIM) standards from the Distributed Management Task Force (DMTF)". I will pass this request along to R&D for consideration for the FireMonkey platform. You should be able to use the Magenta Systems non-visual controls for your FireMonkey Windows applications. Maybe Magenta Systems will consider building the same components for Mac. You might suggest this to them directly.

Q: How does the FireMonkey fit in the new Windows 8 application platform?
A: We have tested RAD Studio XE2 and FireMonkey applications using the Windows 8 preview release and the IDE, tools and compile applications work. R&D has Windows 8 SDK and WinRT on our roadmap for future releases. We will make sure that you can build applications for Windows, Mac, iOS and other platforms using our compilers and one codebase for years to come.

Q: Will we have any lesson about using of built in objects and funtions of iOS?
A: I have been covering a little bit about using FireMonkey on iOS with Delphi in each of the lessons where it is appropriate. I'll think about adding a lesson specific to iOS when we have our next generation compilers that will support iOS and Android directly. In the meantime, Anders Ohlsson's blog contains multiple articles and examples showing how to access iOS from your FireMonkey for iOS applications - http://blogs.embarcadero.com/ao/category/ios. We definitely plan to enhance FireMonkey in the future to fully support iOS and Android devices and the hardware and software included with the devices. John Thomas (JT), EMbarcadero Director of Product Management for Developer Tools, in his recent roadmap article "Coming soon to a RAD IDE near you, the future of C++ - 64bit, C++11, ARM, iOS and Android" located at http://edn.embarcadero.com/article/42275: "The FireMonkey framework is also being updated to fully support C++ iOS and Android mobile development with high fidelity native and custom UIs and native platform services and sensors such as GPS, camera, accelerometers, and more." JT's statement, even though it was stated in the context for future releases of C++Builder, it also holds true for future releases of Delphi.

Q: Do you have a preference of using IFDEF or TOSVersion. I know there are situations that lean toward using one or the other, but, overall, which method do you prefer?
A: I have no specific preference for one or another. I believe it all depends on your application and what it is trying to do in code for multi-platform support. There is a lot of use of ifdef(s) in FireMonkey and RTL source code for platform and compiler specific functionality. In the lesson, I wanted to make sure

you knew about the TOSVersion record's existence and how to use it if you wanted to have code that needed to detect, at run-time, what platform it was running on.

Q: Why would you use Terminate() instead of Close() in a C++ application?
A: You can use either method depending on your program design and logic. In a FireMonkey application, using Application->Terminate() ends the execution of the FireMonkey application from wherever it is called. By calling Terminate rather than freeing the application object, you allow the application to shut down in an orderly fashion. Terminate performs an orderly shutdown of the application and Terminate is not immediate. Terminate() is also called automatically when the user closes the main application form or calls the Close() method for the main form.

If you have any additional questions – send me an email - davidi@embarcadero.com

Self Check Quiz

1. What does the acronym "OOP" stand for?

a) I made a mistake
b) I did it again
c) Objects, Options and Programming
d) Object Oriented Programming

2. A new FireMonkey HD application project includes what files?

a) Main program
b) Main program and a main form
c) Main form only
d) Main program, main form and unit source code

3. Which section is not part of a class declaration?

a) Public
b) Private
c) Protected
d) Protracted
e) Published

4. A constructor is a special method that creates and initializes instance objects, true or false?

a) True
b) False

Answers to the Self Check Quiz:

1d, 2d, 3d, 4a