About the Book

FreeBASIC Beginners Guide is a concise, hands-on beginner's guide to FreeBASIC and is aimed at the novice programmer. The book assumes no prior programming knowledge. The goal of the book is to create a solid foundation in the basics of programming in FreeBASIC that the programmer can build and expand upon.

FreeBASIC is a cross-platform compiler, however this book concentrates on using FreeBASIC under the 32-bit Microsoft Windows™ and GNU/Linux operating systems. However, other than the installation and setup portions of this book, most of the code should run under any of the supported platforms.

All of the source code presented in this book is original code, and can be downloaded from the Github repository at https://https://github.com/patrickw99/fbbiginner

Conventions Used in the Book

The various styles used in the book along with their meaning, are listed below. The basic paragraph format is the font you see here with each paragraph indented. Information that is supplemental to the topic being discussed will be presented in the format below.

These colored boxes have additional, supplemental or historical information associated with the current topic being discussed.

Source code is listed in a fixed font with borders to delimit the code segment. You should always type in the programs. Most are small and can be entered into the editor in just a couple of minutes. The physical act of typing in the programs will help you learn the vocabulary and structure of FreeBasic. The file name for the program is located after the code segment.

```
Cls
Print "Hello World From FreeBasic!"
Sleep
End
```

Listing 1.1: helloworld.bas

Following each listing is an analysis paragraph which explains how the program works It is marked by the word Analysis: in bold. Even though many of the keywords you will see may have not be discussed in detail in the text, seeing these keywords in action will help you become familiar with the vocabulary of FreeBasic.

Analysis: This will mark the beginning of an analysis section. The code will be discussed in this section. Since there will be no line numbers you will need to read the code carefully.

The output of a program is listed in a fixed font and enclosed in a gray box as shown below. This output is given so you can compare the results to see if you have entered the program correctly.

```
Hello World from FreeBasic!
```

Output 1.1: HelloWorld.bas

A potential problem is shown with a Caution paragraph.

Caution: This style will alert you to potential problems and offer some remedies or tips to avoid them.

If we look at the fact, we shall find that the great inventions of the age are not, with us at least, always produced in universities.

Charles Babbage

Table of Contents

A Brief Introduction to FreeBASIC

FreeBASIC is a 32/64-bit BASIC compiler that outputs native code for Microsoft Windows, Linux and a 32-bit BASIC compiler for DOS via DJGPP. FreeBASIC is also Open Source, which means anyone may freely view and edit the source to suit their needs.

Key Features of FreeBASIC

Built-in Graphics library

- Completely compatible with old QB Graphics commands, but it builds on this to offer much more.
- Support for high resolutions and any color depth, and any number of off screen pages.
- All drawing functions can operate on screen as well as on off screen surfaces(GET/PUT buffers) of any size.
- Advanced sprites handling, with clipping, transparency, alpha and custom blending.
- Direct access to screen memory.
- BMP loading/saving capabilities.
- OpenGL support: init an OpenGL mode with a single command, then use GL instructions directly.
- Keyboard, mouse and joystick handling functions.
- The Graphics library is fast: MMX optimized routines are used if MMX is available.
- Small footprint: when using Graphics commands, your EXEs will grow in size by a minimum of 30/40K only.
- Stand-aloness: generated EXEs will depend only upon system libs, no external libraries required.
- As all of FB, also gfxlib is completely multi platform: underneath it uses DirectX or GDI (if DX is not available) under Win32, direct VGA/ModeX/VESA access under DOS, or raw Xlib under Linux.
  Create OBJ's, LIB's, DLL's, and console or GUI EXE's
- You are in no way locked to an IDE or editor of any kind.
- You can create static and dynamic libraries adding just one command-line option (-lib or -dll).

Debugging support

- Full debugging support with GDB (the GNU debugger) or Insight (the GDB GUI front end)

- Array bounds checking (only enabled by the -exx command-line option)
- Null pointers checking (same as above)

Function overloading

- DECLARE SUB Test OVERLOAD (a AS DOUBLE)
- DECLARE SUB Test (a AS SINGLE)
- DECLARE SUB Test (a AS INTEGER, b AS INTEGER = 1234)
- DECLARE SUB Test (a AS BYTE, b AS SHORT)

Inline Assembly

- Intel syntax.
- Reference variables directly by name with no "trick code" needed.

Most of the known C libraries can be used directly

- GTK+ 2.0: cross-platform GUI Toolkit (over 1MB of headers, including support for Glade, libart and glGtk)
- libxml and libxslt: defacto standard XML and XSL libraries
- GSL - GNU Scientific library: complex numbers, vectors and matrices, FFT linear algebra, statistics, sorting, differential equations, and a dozen other sub-libraries with mathematical routines
- GMP - GNU Multiple Precision Arithmetic Library: known as the fastest bignum library
- SDL - Simple Direct Media Layer: multimedia library for audio, user input, 3Dand 2D gfx (including the sub-libraries such as SDL_Net, SDL_TTF, etc)
- OpenGL: portable library for developing interactive 2D and 3D graphics games and applications (including support for frameworks such as GLUT and GLFW)
- Allegro: game programming library (graphics, sounds, player input, etc)
- GD, DevIL, FreeImage, GRX and other graphic-related libraries
- OpenAL, Fmod, BASS: 2D and 3D sound systems, including support for mod,
- mp3, ogg, etc
- ODE and Newton - dynamics engines: libraries for simulating rigid body dynamics
- cgi-util and Fast CGI: web development
- DirectX and the Windows API - the most complete headers set between the Basic compilers available, including support for the Unicode functions
- DispHelper - COM IDispatch interfaces made easy
- And many more!
  Support for numeric (integer or floating-point) and strings types
- DECLARE SUB Test(a AS DOUBLE = 12.345, BYVAL b AS BYTE = 255, BYVAL sAS STRING = "abc")

Unicode support

- Besides ASCII files with Unicode escape sequences (\u), FreeBASIC can parse UTF-8, UTF-16LE, UTF-16BE, UTF-32LE and UTF-32BE source (.bas) or header(.bi) files, they can freely mixed with other sources/headers in the same project (also with other ASCII files).
- Literal strings can be typed in the original non-latin alphabet, just use an text-editor that supports some of the Unicode formats listed above.
- The WSTRING type holds wide-characters, all string functions (like LEFT,TRIM, etc) will work with wide-strings too.

Unlimited number of symbols

- Being a 32/64-bit application, FreeBASIC can compile source code files up to 2GB or loarger.
- The number of symbols (variables, constants, et cetera) is only limited by the total memory available during compile time. You can, for example, include OpenGL, SDL, BASS, and GTK simultaneously in your source code.

Section 1 - Introduction to FreeBasic

Chapter 1 - FreeBASIC Data Types

Introduction to Data Types

When starting out with a new programming language, one of the first things you should learn is the language's data types. Virtually every program manipulates data, and to correctly manipulate that data, you must thoroughly understand the available data types. Data type errors rank second to syntax errors, but they are a lot more troublesome. The compiler can catch syntax errors and some data type errors, but most data type errors occur when the program is running, and often only when using certain types of data. This kind of intermittent errors is difficult to find and fix. Knowing the kind, size and limits of the data types will help keep these kinds of errors to a minimum.

FreeBASIC has all the standard data types that you would expect for a Basic compiler, as well as pointers which you usually only find in lower-level languages such as C. Table xx lists the basic different data types that FreeBASIC supports. Each Data type will be examined more in depth in the their own chapter.

Datatype	Description
Numeric	Includes both singed and unsigned floating point numbers and integers.
String	Includes fixed-length and dynamic strings. Also includes NULL terminated strings.
Pointers	A pointer contains the memory address of data.
Arrays	Arrays are contiguous memory segments of a single or composite data type.
Composite	A user defined data type based on other data types.

Table 1.1: FreeBASIC Data Types

Variables

Variables are names which can be manipulated. They are declared and referenced using the Dim statement. A variable is a name composed of letters, numbers or the underscore character such as MyInteger, or MyInteger2. There are rules for variable naming that you must keep in mind.

1. Variable names must start with a letter or the underscore character. It is not recommended however, to use the underscore character, as this is generally used to indicate a system variable, and it is best to avoid it in case there may be a conflict with existing system variable names.
2. Most punctuation marks have a special meaning in FreeBASIC and cannot be used in a variable name. While the period can be used, it is also used as a deference operator in Types, and so should not be used to avoid confusion or potential problems with future versions of the compiler.
3. Numbers can be used in a variable name, but cannot be the first character. MyVar1 is legal, but 1MyVar will generate a compiler error.

To create a variable, it first must be defined. All variables must be dimensioned before they can be used.

The Dim Statement

To declare variables in your program, you use the Dim statement. The Dim statement instructs the compiler to set aside some memory for a variable of a particular data type. For example, the statement **Dim myInteger as Integer** instructs the compiler to set aside 4 byes of storage. Whenever you manipulate myInteger in a program, the compiler will manipulate the data at the memory location set aside. Just as with variable names, there are some rules for the use of Dim as well. The table below shows the four basic forms of the DIM statement.

Example	Discription
Dim myVar As {Data Type} Dim myVar As {Data Type}, myVar2 As {Data Type}, …	This will create one or more variables of the given DataType.
Dim As {Data Type} myVar, myVar2, ...	This will create two or more variables of the same Datatype.
Dim myVar As {DataType} = {Value}	This will create a variable of the given DataType and set the value
Dim myVar as {DataType} ptr	This will create a pointer to a given Datatype

Table 1.2: Dim Statement

Caution: Dim myVar, myVar2 As Double. This may look like it creates two double-type variables, however myVar will not be defined and will result in compilation errors. Use rule 2 if you want to create multiple variables of the same type.

Brief Introduction to Variable Scope.

When a variable is created, you can define how it can be accessed. This is called Variable Scope. Variable scope refers to how and when a variable can be accessed. Or in other words it's visibility to the rest of the program. When a variable is defined it can be modified by using the following keywords.

Keyword	Example	Description
Shared	Dim Shared myVar As {Data Type}	can be accessed anywhere within the program's current module
Static	Dim Static myVar As {Data Type}	Static variables are used within subroutines and functions and retain their values between calls.
Common	Dim Common myVar As {Data Type}	shared between multiple code modules.
Extern and Import	Dim Extern myVar As {Data Type} Dim Import myVar As {Data Type}	used when creating DLL's

Table 1.3: Variable Scope Types

The above modifiers define where a variable can be accessed. Next is a general overview of variable scope. Each of these are discussed in more detail latter in the book.

Shared Variables

As you can see in the table above, using the Dim Shared creates a shared variable. This means that any variable you create as Shared can be accessed anywhere within the program's current module or .bas file. The number of variables in your program (as well as the program itself) is limited only by your available memory, up to 2 gigabytes for a 32bit system.

Static Variables

Static variables are used within subroutines and functions and retain their values between calls. Static variables can only be defined within a subroutine or function. Variables declared outside of a subroutine or function, that is at the module level, will maintain their values and are static by default. Subroutines and Functions will be discussed in detail later in this book.

Common Variables

Variables declared as Common can be shared between multiple code modules, that is between multiple .bas files in the same program. Common variables will be discussed in detail later in this book.

Extern and Import Variables

Extern and Import are used when creating DLL's and like Common, are designed to share variables in different modules. Extern and Import are beyond the scope of the book. More information can be found on the FreeBASIC website.

Which Data Type To Use?

There are a number of different data types available, so how do you choose the right data type for any given application? The rule of thumb is to use the largest data type you need to hold the expected range of values. This may seem like stating the obvious, but many programs fail because the programmer did not fully understand the range of data in their program. When you create a program, you should map out not only the logic of the program,

but the data associated with each block of logic. When you map out the data ahead of time, you are less likely to run into data-type errors.

For example, if you were working with ASCII codes, which range from 0 to 255, an ubyte would be a good choice since the range of an ubyte is the same as the range of ASCII codes, and you are only using 1 byte of memory. There is another consideration though, the "natural" data size of the computer. On a 32-bit system, the natural data size is 4 bytes, or an integer. This means that the computer is optimized to handle an integer, and does so more efficiently, even though you are "wasting" 3 bytes of memory by using an integer for an ASCII code.

In most cases an integer is a good general-purpose choice for numerical data. The range is quite large, it handles both negative and positive numbers and you benefit from using the computer's natural data type. For floating point data, a double is a good choice since, like the integer, it has a good range of values and better precision than a single. For text data you should use a string or Zstring. These are only suggestions; what data type you end up using will be dictated by the needs of your program.

These "rules of thumb" apply to single variable declarations where a few wasted bytes are not critical. However, as you will see in the chapter on arrays, choosing the rightsized data type is critical in large arrays where a few wasted bytes can add up to large amount of wasted memory.

Chapter 2 - Numeric Data Types

FreeBASIC has all the standard Numeric data types that you would expect for a Basic compiler. See Table 2.1 for a list of supported numeric data types. You will notice that Integer and Long are grouped together. This is because a Long is just an alias for Integer. They are exactly the same data type.

Numeric Data	Types Size	Limits
Byte	8-bit, signed, 1 byte	-128 to 127
Integer	(Long) 32-bit, signed, 4 bytes	-2,147,483,648 to 2,147,483,647
LongInt	64-bit, signed, 8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
Short	16-bit, signed, 2 bytes	-32,768 to 32,767
UByte	8-bit, unsigned, 1 byte	0 to 255
UInteger	32-bit, unsigned, 4 bytes	0 to 4,294,967,295
ULongInt	64-bit, unsigned, 8 bytes	0 to 18,446,744,073,709, 551,615
Ushort	16-bit, unsigned, 2 bytes	0 to 65365
Double	64-bit, floating point, 8 bytes	-2.2E-308 to +1.7E+308
Single	32-bit, floating point, 4 bytes	1.1 E-38 to 3.43 E+38

Table 2.1: FreeBASIC Numeric Data Types

Signed Versus Unsigned Data Types

Signed data types, as the name implies, can be negative, zero or positive. Unsigned types can only be zero or positive, which give them a greater positive range than their signed counterparts. If your data will never be negative, using the unsigned data types will allow you to store larger numbers in the same size variable.

The Floating Point Data Type

The floating point data types, Single and Double, are able to store numbers with decimal digits. Keep in mind that floating-point numbers are subject to rounding errors, which can accumulate over long calculations. You should carry more than the number of decimal digits than you need to ensure the greatest accuracy.

Using Different Number Formats

Besides decimal numbers, FreeBASIC is able to recognize numbers in hexadecimal, binary and octal formats. Table 3.4 lists the number base and format to use.

Number Base	Format
Decimal	myVar =254
Hexadecimal	myVar =&HFE
Binary	myVar =&B11111110
Octal	myVar =&O376
Exponential Numbers	myVar =243E10

Table 2.2 Format of Number Bases

Hexadecimal Numbers

Hexadecimal is a base 16 numbering scheme and have digits in the range of 0 to F. Hexadecimal numbers are commonly used as constant values in the Windows API and many third party libraries as it is a compact way to represent a large value. To indicate a hexadecimal number, use the &H prefix.

Binary Numbers

Binary is a base 2 numbering scheme and have digits in the range of 0 and 1. Binary is the language of the computer. Although we can enter numbers and letters into the computer, it all must be translated into binary before the computer can understand it. To indicate a binary number, use the &B prefix.

Octal Numbers

Octal is a base eight numbering scheme and have digits in the range of 0 to 7. Octal numbers were very popular in early computer systems, but aren’t used much today except in some specialized applications. To indicate an octal number, use the &O prefix.

Exponential Numbers

You can use exponential numbers in your program by adding the E suffix followed by the power. To use the number 105, you would write the number as 10E05. You can directly set a double or single type variable using the exponent format. You can also use negative exponents such as 10E-5, which when printed to the screen would like 1.e004.

Exercises

Here are some questions to test what you have read.

1) What data type would be the best to store the number 196?
2) What data type would be the best to store the number 2.134?
3) What data type is the best for general usage on 32-bit systems?
4) What is the difference between signed and unsigned data types?
5) What prefix would you use to designate a binary number?
6) What prefix would you use to designate a hexadecimal number?
7) What Alphabetic letters are allowed in a hexadecimal number?
8) What would the hexadecimal number 1AF be in decimal form?

Chapter 3 - Arithmetic Operators

Arithmetic Operators and Precedence

Arithmetic operators are used to perform mathematical calculations. The Table below lists all the FreeBasic arithmetic operators in order of precedence. The square brackets in the syntax column indicate optional additional expressions.

Function	Syntax	Comment
( ) (Parenthesis)	B = expression operator (expression [operator expression [(…])	Forces evaluation of expression. Parenthesis can be nested, but must be closed properly or a compile error will occur.
^ (Exponentiation)	B = expression^value	Returns the result of raising expression to the power of value. That is 2^2 = 2*2.
- (Negation)	B = - expression	Returns the negated value of expression. This is the same as multiplying by –1. If expression is positive, B will negative. If expression is negative, B will positive.
* (Multiplication)	B = expression * expression[* expression…]	Returns the product of two or more expressions.
/ (Division)	B = expression / expression[/ expression…]	Returns the division of two or more expression. The result implicitly converted to the target data type. That is, if B is an integer, the result will be rounded, if B is a double or single, the result will be a decimal number. Note: If the second expression

Function	Syntax	Comment
		evaluates to zero(0), then a run time error will occur.
\ (Integer Division)	B = expression \ expression[\ expression…]	Returns the integer result of division of two or more expressions. The result is implicitly converted to an integer. Note: If the second expression evaluates to zero(0), then a run time error will occur
MOD (Modulo)	B = expression MOD	Returns the remainder of expression the implicit division of the expressions as an integer result. If expression is a decimal value, expression is rounded before the division.
+ (Addition)	B = expression + expression [+ expression…]	Returns the sum of two or more expressions. Expression can be a number, variable or another expression.
- (Subtraction)	B = expression – expression[- expression…]	Returns the difference between two or more expressions.

Table 3.1: Arithmetic Operators in Order of Precedence

You should be familiar with most of these operators from math class. There are three operators that you may not be familiar with. These are

- Integer division: Used when you are not concerned about the remainder of the division process.
- SHL and SHR operators: They will be discussed in the section on Bitwise Operators. They are included here to show where they fall in the precedence rules.
- Mod operator: Returns the remainder part of the division of two or more expressions as an integer.

When using these operators together in single statement, you must b aware of how FreeBasic evaluates the expression. For example, does `5 + 6 * 3 equal 33` or does it equal 23? FreeBasic evaluates expressions based on precedence rules, that is, rules that describe what gets evaluated when. The table above lists the arithmetic operators in order of precedence with the following exceptions.

- Multiplication and division have the same precedence.
- SHL, SHR have the same precedence.
- Addition and Subtraction have the same precedence.

Looking at the table and the equation 5 + 6 * 3 you can see that this will evaluate to 23 not 33, since multiplication has a higher precedence then addition. The compiler will read the expression from left to right, pushing values onto an internal stack until it can resolve part or all of the equation. For this equation 5 will be read and pushed, then the + operator will be read and pushed onto the stack. Since the + operator requires two operands, the compiler will read the next element of the expression which will be the * operator. This operator also requires two operands, so * will be pushed onto the stack and the 3 will be read. Since * takes priority over +, the 6 and 3 will be multiplied, and that value will be stored on the stack. At this point there is a 5, + and 18 on the stack. Since there is only one operator left and two operands, the 5 and 18 will be added together to make 23, which will be returned as the result of the expression.

If you wanted to make sure that 5 + 6 gets evaluated first, then you would use parenthesis to force the evaluation. You would write the parenthesized expression as (5 + 6) * 3. Since the parenthesis have the highest precedence, the expression they contain will be evaluated before the multiplication. The evaluation process here is the same as the previous process. The (is treated as an operator and is pushed onto the stack. The 5, +, and 6 are read followed by the). The) signals the end of the parenthesis operation, so the 5 and 6 are added together and pushed onto the stack. The * is read along with the 3. The stack at this point contains an 11, * and 3. The 11 and 3 are multiplied together and33 is returned as the result of the evaluation.

You can also embed parenthesis within parenthesis. Each time the compiler encounters a (, it begins a new parenthesis operation. When it encounters a), the last (issued to evaluate the items contained within the (and) and that result is either placed on the stack for further evaluation or returned as a result. Each (must be match to a) or a compiler error will occur.

The following program demonstrates both implicit evaluation and force devaluation.

```
Dim As Integer myInt

'Let compiler evaluate expression according to precedence
```

```
myInt = 5 + 6 * 3
Print "Expression 5 + 6 * 3 = ";myInt

'Force evaluation using parenthesis
myInt = (5 + 6) * 3
Print "Expression (5 + 6) * 3 = ";myInt

'Wait for key press

Sleep
 End
```

Listing 3.1: precedence.bas

Analysis: As always we open the program with declaring the working variables. The compiler evaluates the first math expression according to precedence rules and stores the result in myInt, which is printed to the console window. The second math expression uses parenthesis to force the evaluation of the expression. Which is then printed to the console screen. The program is then closed usual way.

As you can see from the output below, using parenthesis allows you to control the evaluation of the expression.

```
Expression 5 + 6 * 3 = 23
Expression (5 + 6) * 3 = 33
```

Output 3.1: precedence.bas

What about the case where two operators are used that have the same precedence level? How does FreeBasic evaluate the expression? To find out, run the following program.

```
Dim As Integer myInt

'Expression 1
myInt = 3 + 5 - 4

Print "Expression 1: 3 + 5 - 4 is";myInt

'Expression 2
myInt = 3 - 5 + 4

Print "Expression 2: 3 - 5 + 4 is";myInt
```

```
 'Expression 3
 myInt = 6 * 2 / 3

 Print "Expression 3: 6 * 2 / 3 is";myInt

 'Expression 4
 myInt = 6 / 2 * 3

 Print "Expression 4: 6 / 2 * 3 is";myInt
 Sleep
End
```

Listing 3.2: sameprecedence.bas

Analysis: The working variable myInt is declared, The first expression has the + operator first and the – second, while Second expression the order has been reversed to test the evaluation order. The results are printed. The the same process is repeated using the * and / operators on the same manner to test the order of evaluation and the results are printed.

Running the program produces the following result.

```
Expression 1: 3 + 5 - 4 is 4
Expression 2: 3 - 5 + 4 is 2
Expression 3: 6 * 2 / 3 is 4
Expression 4: 6 / 2 * 3 is 9
```

Output 3.2: Output of sameprecedence.bas

As you can see from the output each expression has been evaluated from left to right. This is the case where parenthesis are very helpful in ensuring that the evaluation order is according to your program needs. When in doubt, always use parenthesis to ensure that the result is what you want in your program.

Shortcut Arithmetic Operators

FreeBasic has a number of shortcut arithmetic operators similar to those found in the C programming language. These operators have the same precedence as their single counterparts. The following table lists the shortcut operators.

Operator	Syntax	Comment
+=	B += expression	Add B to expression and assign to B.
-=	B -= expression	Subtract B to expression and assign to B.
*=	B *= expression	Multiply B to expression and assign to B.
/=	B /= expression	Divide B by expression and assign to B.
\=	B \= expression	Integer divide B by expression and assign to B.

Table 3.3: Shortcut Arithmetic Operators

Here is a simple program that Shows how shortcut operators work.

```
 Dim As Integer myInt
myint = 10
Print " myint = "; myint
'Expression 1
myInt += 2

Print "Expression 1: myInt += 2 is";myInt

'Expression 2
myInt -= 2

Print "Expression 2: myInt -= 2 is";myInt

'Expression 3
myInt *=  4

Print "Expression 3: myInt *= 4 is";myInt

'Expression 4
myInt  /=  4

Print "Expression 4: myInt /= 4 is";myInt
Sleep
End
```

Listing 3.2: shortcut.bas

Analysis: The program starts by defining the variable myint and then assigning it the value of 10. Then next several sections add, subtract, multiply, and divide myint and print out the results of each operation. Here is the output from this program.

```
  myint =  10
Expression 1: myInt += 2 is 12
Expression 2: myInt -= 2 is 10
Expression 3: myInt *= 4 is 40
Expression 4: myInt /= 4 is 10
```

Using these operators will cut down on the typing you have to do, especially for statements such as `a = a + 1`, which can be written as `a += 1`.

Chapter 4 - String Data Types

Introduction to Strings

Strings are an essential part of programming. Proper manipulation of strings is an essential skill for any programmer. Strings are used in virtually every program you will write, from games to data entry programs. Strings are important because humans use strings to communicate concepts and ideas. A menu in a game communicates the various game options to the user. An address stored in a database communicates the location of the person who lives at that address. Programs a resolutions to problems, and in most cases, must communicate that solution to the user through the use of strings.

Computers however, only understand numbers. The string “¡FreeBasic estáfresco!“ has no meaning to the computer. For the very first computers this did not matter much, since those early computers were just glorified calculators, albeit calculators that took up a whole room. However it soon became evident to computer engineers that in order for the computer to be a useful tool, it needed to be able to somehow recognize string data, and be able to manipulate that string data.

Since computers only understand numbers, the solution was to convert alphanumeric characters to numbers via translation tables. The familiar ASCII code table is one such translation scheme that takes a set of alpha-numeric characters and converts them to numbers. The “A” character is encoded as decimal 65, the exclamation point as decimal 33 and the number 1 is encoded as decimal 49. When you press any key on your keyboard, a scan code for that key is generated and stored in the computer as a number.

Humans group letters together to form words, and words to form sentences. A certain arrangement of letters and words mean something according to the semantic rules the writer's language. When you read the string “FreeBasic is cool!” you understand the meaning of the words if you can read the English language. The computer however doesn't know what FreeBasic is and doesn't know what it means to be cool. When you read that something is “cool” you know that this is slang for great or excellent. All the computer can do is store the string in memory in a way that preserves the format of the string so that a human reading the string will understand its meaning.

While computers have grown in both speed and capacity, the basic computing processes haven't changed since the 1950's. The next revolution in computing won't be quantum processors or holographic memory; it will be when computers can understand language.

The solution to the storage problem was to simply store string data in memory as a sequence of bytes, and terminate that sequence of bytes with a character 0. To put it another way, a string in computer memory is an array of characters, terminated with a character 0 to signal the end of the string. Virtually all early programming languages, and many modern ones, have no native String data type. A string in C is a Null (character 0) terminated array of Char, where Char can be interpreted as both a character and number.

While this scheme accurately reflects the internal structure of a string in memory, it is hard to work with and error prone, and doesn't reflect the way humans work with string data. A better solution to the problem was the creation of the native String data type. The internal structure of a string has not changed, it is still stored in memory as a sequence of bytes terminated by a Null character, but the programmer can interact with string data in a more natural, humanistic way.

FreeBasic has four intrinsic string data types, listed in the following table.

String Type	Declaration	Purpose
Dynamic String	Dim myString as String	8-bit string that is dynamically allocated and automatically resized.
Fixed Length String	Dim myString as String * length	8-bit, fixed-length string.
Zstring	Dim myString as Zstring * length Dim myString as Zstring Ptr	8-bit, fixed-length, Null terminated character string. A Zstring is a C-style string.
Wstring	Dim myString as Wstring *length Dim myString as Wstring Ptr	16-bit, fixed-length, Null terminated string used with Unicode functions.

Table 4.1: Intrinsic FreeBasic String Data Types

All FreeBasic strings can hold up to 2 GB of data, or up to the amount of memory on your computer.

Dynamic Strings

Dynamic strings are variable-length strings that the compiler automatically allocates and resizes as needed. Dynamic strings are actually structures stored in memory, called a string descriptor, that contain a pointer to the string data, the length of the string data and the size of the string allocation. Dynamic strings are allocated in 36byte blocks, which reduces the amount of resizing that the compiler must do if the string size changes. The following program will show an example of dynamic strings at work.

```
'Define some Dynamic strings
Dim MyString As String
Dim MyString2 as String

MyString = "Hello" ' assign a value to the string and print
it out
Print MyString

MyString += ", world!" ' Append a second string to the
first.
Print Mystring

' Assign a value to the second string and concatenate them
' together.
Mystring2 = "Welcome to FreeBASIC"
Print MyString + "! " + Mystring2 + "!"

Sleep
end
```

Listing 4.1: DynamicStr.bas

When you run the program you should see the following output.

```
Hello
Hello, world!
Hello, world!! Welcome to FreeBASIC!
```

Output 4.1: Output of DynamicStr.bas

Dynamic strings use Zstrings strings internally so that you can pass a dynamic string to a function that expects a Null terminated string and the function will work correctly. The Windows API, the C run time library and most third-party libraries written in C expect a Null terminated string as a string parameter.

Caution: In other versions of Basic, strings have been used to load and manipulate binary data. If the binary data contains a Null character, this could cause problems with FreeBasic strings, since the Null character is used to terminate a string. While it is possible to have an embedded Null in a dynamic string (since dynamic strings have a string descriptor), if you pass this data to a third party library function that is expecting a C-style string, the data will read only up to the Null character, resulting in data loss. Instead of using strings to read binary data, you should byte arrays instead. Since a dynamic string is actually a pointer to a string descriptor, you cannot use dynamic strings in type definitions that you are going to save to a file. You should use a fixed length string instead.

Fixed Length Strings

Fixed length strings are defined using a length parameter, and can only hold strings that are less than or equal to the defined size. Trying to initialize a fixed length string with data that is longer than the defined size will result in the string being truncated to fit.

The following short program illustrates creating and using a fixed length string.

```
'Define a fixed length string that can hold up to 20
characters
Dim myFString As String * 20

'Save some data in the string
myFString = "This should fit."
Print myFString '  String is shorter than 20 Characters, the
              '  whole string is printed

'This data will be truncated.
myFString = "This string will be truncated."
Print myFString    ' String is longer than 20 characters, the
                   ' String will be truncated.
Sleep
End
```

Listing 4.2: fixedstr.bas

When you run the program you should see the following output.

```
This should fit.
This string will be
```

Output 4.2: Output of fixedstr.bas

As you can see from the output, if a string is too long to fit in the variable, it will be truncated to fit. Fixed length strings are very useful when creating random access records using type definitions. This technique is used in the chapter on file handling.

Zstrings

Zstrings are Null terminated, C-style strings. The main purpose of a Zstring is to interface with third-party libraries that expect a C-style strings, however they are useful even if you do not plan to pass them to third-party libraries. Zstrings can be defined just like a fixed-length string, Dim myZstring as Zstring * 10, and FreeBasic will handle them just like fixed strings, automatically truncating data to fit the defined size.

Unlike fixed strings however, Zstrings can be dynamically managed by declaring a Zstring pointer and using the associated memory functions. When using a dynamically allocated Zstring, you must be careful not to overwrite the size of the string, as this will overwrite parts of memory not contained in the string and may cause the program or even the operating system to crash.

When using either type of Zstring, FreeBasic will manage the terminating Null character for you, but the storage size of a Zstring will be 1 less than the defined size since the Null character occupies the last character position. When calculating the size of a Zstring be sure to add 1 to the value to account for the Null terminator. Zstrings will be discussed in more detail in the chapter on using the C runtime Library.

Wstrings

Dynamic, fixed and Zstrings all use 1 byte per character. Wstrings, also called wide strings, use 2 bytes per character and is generally used in conjunction with Unicode strings functions. Unicode is, strictly speaking, a character coding scheme designed to associate a number for every character of every language used in the world today, as well as some ancient languages that are of historical interest. In the context of developing an application, Unicode is used to internationalize a program so that the end-user can view the program in their native language. Wstrings can be both fixed length and dynamic and are similar to Zstrings. Wstrings are beyond the scope of this guide. More information and examples can be found on the FreeBasic website.

Chapter 5 - Arrays

Introduction to Arrays

Arrays are probably the single most useful programming construct that is available to you in FreeBasic. Many problems that you will try to solve with a programming solution involve data arranged in tabular format, and arrays are perfect for managing this type of data. Understanding arrays is crucial skill in becoming a competent programmer.

Arrays are contiguous memory segments of a single or composite data type. You can think of an array as a table, with rows and columns of data. An array can have one or more rows, and each row can have one or columns. The number of rows and columns define the dimensions of the array. FreeBasic uses the row-major scheme for arrays, which means that the first dimension references the row in an array that has more than one dimension. FreeBasic supports up to eight dimensions in an array.

One-Dimensional Arrays

An array with a single row is called a one-dimensional array. If an array is a single-dimensional array, then the row is not defined in the declaration, only the number of columns in the row. Since an array requires a minimum of one row, the row is understood to exist in this case. The following code snippets create a single-dimension integer array using the different array definition schemes available in FreeBasic.

```
Dim myArray(10) as Integer
```

This will define an array with a single row and 11 columns, with column indexes(numbers) ranging from 0 to 10. The base array index is 0 if the lower bound of the array is not defined. This behavior can be changed using the Option Base n compiler directive. Setting Option Base 1 with the above example would result in an array with 10columns, with the indexes ranging from 1 to 10. The Option Base directive must be defined before dimensioning any arrays.

```
Dim myArray(1 to 10) as Integer
```

This example will define a single-dimension array with 10 columns, with indexes ranging from 1 to 10.

One-Dimensional Array Indexes

You access each element of an array using an index value. In the case of a single-dimension array, the index would refer to a column number in the default row. The format is to use the array variable, with the index surrounded by parenthesis.

This would set the value of column 5 of myArray to 7.

```
myArray(5) = 7
```

This will set the value of myInt to the current value of column 5 in myArray.

```
myInt = myArray(5)
```

Two-Dimensional Arrays

A two-dimensional array is an array that has more than one row, along with the defined columns. A two-dimensional array is like a table, with a defined number of rows, where each row has a defined number of columns. The following code snippet defined an array using the default method.

```
Dim myArray(2, 10) as Integer
```

The first dimension defines the number of rows in the array, while the second dimension defines the number of columns in each row. In this example, the array has 3 rows, numbered 0 to 2, and each row has 11 columns, numbered 0 to 10.

You can also define the lower and upper bounds of the array.

```
Dim myArray(1 to 2, 1 to 10) as Integer
```

This definition would set the number of rows to 2, numbered 1 to 2 and the number of columns to 10, numbered 1 to 10.

Two-Dimensional Array Indexes

To access the array elements you would use two indexes. The first index selects the row, and the second index selects a column within that row.

This code would set column 5 in row 1 to 7.

```
myArray(1, 5) = 7
```

This code would set myInt to the current value contained within column 5 of row 1of myArray.

```
myInt = myArray(1, 5)
```

The following program illustrates creating and accessing a two dimensional array.

```
'Create a two dimensional array and to integer variables
Dim As Integer myArray(1 To 2, 1 To 10), i, j

'Load some data into the array
Randomize (Timer)

For i = 1 To 2          'loop for first index’
    For j = 1 To 9           'loop for second index.
         myArray(i, j) = Rnd * 10
    Next
Next

'Print data in array
For i = 1 To 2          'loop for first index.
    For j = 1 To 9           'loop for second index.
         Print "row:";i;" col:";j;" value:";_
         myArray(i, j)
    Next
Next

Sleep
End
```

Listing 5.1: arrayindex.bas

Analysis: A two-dimensional array is created with two rows, with each row having ten columns. The working variables i and j are also declared on the same line. A nested For-Next loop is used to load some random data into the array. This method is the common way to access multi-dimensional arrays. The variable i will select the row indexes while the j variable will access the column indexes. Another nested For-Next loop is used to print the values stored in the array. The format is identical to the load code. The program is closed in the usual way.

When you run the program you should see the following output.

```
 row: 1 col: 1 value: 0
 row: 1 col: 2 value: 4
 row: 1 col: 3 value: 4
 row: 1 col: 4 value: 1
 row: 1 col: 5 value: 9
 row: 1 col: 6 value: 7
```

```
row: 1 col: 7 value: 4
row: 1 col: 8 value: 1
row: 1 col: 9 value: 5
row: 2 col: 1 value: 10
row: 2 col: 2 value: 10
row: 2 col: 3 value: 5
row: 2 col: 4 value: 9
row: 2 col: 5 value: 9
row: 2 col: 6 value: 5
row: 2 col: 7 value: 3
row: 2 col: 8 value: 8
row: 2 col: 9 value: 4
```

Output 5.1: arrayindex.bas

Multi-Dimensional Arrays

For arrays of three or more dimensions, you would use the same format as listed above, taking into account the progression of the array dimensions. For a three-dimensional array, the first dimension would be the row, the second the column, the third would be the z-order, or depth, of each column. For example, to define a cube in space, you would use the y,x,z format, where y defines the vertical axis, x defines the horizontal axis and z defines the depth axis. To create an array in this format you could define the array as Dim myCube(y, x, z) as Integer. MyCube(10, 10, 10) would create a cube with 11 vertical units, 0 to 10, 11 horizontal units, 0 to 10 and 10 depth units, 0 to 10. To access the center of the cube, you would use iCenter = myCube(5, 5, 5).

You will probably never need to use arrays of more than three dimensions, unless you are doing some advanced mathematical calculations. However, if you need to use higher-dimensional arrays, the same principles apply.

Dynamic Arrays

The arrays described above are static arrays; the array size cannot change during program execution. You can also create dynamic arrays that can change size during execution. Dynamic arrays are useful for creating data structures such as stacks or queues.

Static arrays, the arrays described above, are kept on the heap, but dynamic arrays are allocated from the computer's pool of memory.. You can dimension a dynamic array in two ways. The first is to declare the array size in the Dim statement and then resize it using Redim or Redim Preserve. The second is to not specify the array size, use empty parenthesis, and then use Redim or Redim Preserve to size the array. Redim will size the array and clear the array contents. Redim Preserve will size the array. and keep any existing data in the array.

There are a couple of exceptions to this. When declaring an array using variables for the index values, the array is implicitly dynamic. You can also declare an array without specifying index values, that is using an empty parentheses in the array declaration, and then use Redim to resize the array.

The following program creates a simple integer stack and then manipulates the stack.

```
'Create an integer stack. The 0 index will be our empty marker.
Dim As Integer stack()
dim I as integer        'loop counter
dim ivalue as integer   'Value to push on stack.
dim stacktop as integer 'Top of stack.

'This will push a value on the stack, update the top of ‘stack
'Push five values on to stack
Print "Pushing values on stack..."
For i = 1 To 5
    ivalue = i * 2
    stacktop = i
    ReDim Preserve stack(stacktop)
    stack(stacktop) = ivalue
    Print "Stack top:";Ubound(stack), "Value:"; stack(stacktop)
Next '

Print
'This will pop a value off the stack, update top of ‘stack
'Pop five values off the stack
Print "Popping values from stack..."

Do While stacktop > 0
    Print "Stack top:";Ubound(stack),"Value:";stack(stacktop)
    If stacktop = 0 Then
         Print "array value = "; stack(stacktop)
    Else
         stacktop -= 1
         ReDim Preserve stack(stacktop)
    End If
Loop
Print

'Check stack size
Print "Stack size after pops:";Ubound(stack)

Sleep
End
```

Listing 5.2: stack.bas

Analysis: The program starts by dimensioning the working variables. The 0 index in the stack array will be used to indicate that the stack is empty. Even though this wastes one integer, it makes the code easier to implement. Next a for-next loop is used to push 5 values onto a stack. The top-of-stack variable, and the value to push on to the stack are initialized . You could determine the top of the stack using the Ubound function, however in a real stack implementation, you will probably need to pass the top-of-stack to other functions, and keeping the value in a variable will reduce the number of calculations the program must do. Each time a new value is pushed the array has a Redim Preserve performed. At the end of the loop we have an array of 5 values.

Next the values are popped from the stack and the array shrunk by 1 using the Redim Preserve again. A check to make sure that the stack has some data. If the index is 0, the stack is empty. Once the array has 0 values the program exits.

In line the Option Dynamic directive is used to create the stack array using dynamic memory. Line 5 dimensions the working variables. The 0 index in the stack array will be used to indicate that the stack is empty. Even though this wastes one integer, it makes the code easier to implement. Line 8 through 14 defines the code to push a value on to the the stack. The parameters of the subroutine are the stack array, the top-of-stack variable, and the value to push on to the stack. You could determine the top of the stack using the Ubound function, however in a real stack implementation, you will probably need to pass the top-of-stack to other functions, and keeping the value in a variable will reduce the number of calculations the program must do. The program is closed in the usual way.

When you run the program you will see the following output.

```
Pushing values on stack...
Stack top: 1  Value: 2
Stack top: 2  Value: 4
Stack top: 3  Value: 6
Stack top: 4  Value: 8
Stack top: 5  Value: 10

Popping values from stack...
Stack top: 5  Value: 10
Stack top: 4  Value: 8
Stack top: 3  Value: 6
Stack top: 2  Value: 4
Stack top: 1  Value: 2

Stack size after pops: 0
```

Output 5.2: stack.bas

Stacks have a wide range of uses, however most implementations use pointers since it is much faster to manipulate a pointer than it is an array. The advantage of an array is the simplicity of use, and the readability of the code, but if you need speed, you should use a pointer memory array for stacks.

You can only Redim or Redim Preserve the first index, that is the row, in a multidimensional array.

Array Functions

There are a number of functions that you can use to manage arrays. The following table lists the array functions available in FreeBasic.

Function	Syntax	Comment
Clear	Clear array, value, num_bytes	Sets num_bytes in array to byte value.
Erase	Erase array	Erases dynamic arrays from memory, or clears static arrays.
Lbound	B = Lbound(array) B = Lbound(array, dimension)	Returns the lowest index of a single or multidimensional array. The dimension parameter is the dimension to check in a multidimensional array where the first dimension is 1, the second dimension is 2, and so on.
Preserve	Redim Preserve	Preserves data in an array when resized using the Redim statement.
Redim	Redim array(dimensions) as DataType	Resizes an array to size dimensions.
Ubound	B = Lbound(array) B = Lbound (array, dimension)	Returns the highest index of a single or multidimensional array. The dimension parameter is the dimension to check in a multidimensional array where the first dimension is 1, the second dimension is 2, and so on.

Table 5.1: Array Functions

Out of all the functions, you will probably use the Lbound and Ubound functions the most, especially when working with dynamic arrays. The following program uses both functions to get to Create a dynamic array

```
'Create a dynamic array
Dim As Integer myArray(), i, nb, row, Column

'Resize array with two dimensions
Redim myArray(1 To 1, 1 To 5)
```

```
'Print upper and lower bounds for each row and column
Print "First Redim"
Print ""
Print "Min Row Index:";Lbound(myArray, 1);
Print " Max Row Index:";Ubound(myArray, 1)
Print "Min Column index:";Lbound(myArray, 2);
Print " Max Column index:";Ubound(myArray, 2)
Print
Print "Additional Redims"
Print ""
'Redim array five times
For i = 1 To 5
    nb = Ubound(myArray, 1) + 1
    Redim myArray(1 To nb, 1 To 5)
    'Print new array size
    Print "Min Row Index:";Lbound(myArray, 1);
    Print " Max Row Index:";Ubound(myArray, 1)
    Print "Min Column index:";Lbound(myArray, 2);
    Print " Max Column index:";Ubound(myArray, 2)
    Print
Next

Sleep
End
```

Listing 5.3: ulbound.bas

Analysis: A dynamic array is created with no dimensions specified. Other Variables are defined The first Redim creates a two-dimensional array. The upper and lower bounds are printed using Lbound and Ubound respectively. Notice that the dimension parameter is used to specify which dimension to get the lower and upper bounds. 1 is used to get the bounds for the first dimension, and 2 is used to get the second dimension.

A For-Next loop is used to resize the first dimension of the array and print the new lower and upper indexes. The current upper bound of the first dimension, has 1 is added to it. Then the value is used with Redim in to resize the first dimension of the array. The new bounds are then printed to the screen. The program is closed in the usual way.

When you run the program you should see the following output.

```
First Redim

Min Row Index: 1 Max Row Index: 1
Min Column index: 1 Max Column index: 5
```

```
Additional Redims

Min Row Index: 1 Max Row Index: 2
Min Column index: 1 Max Column index: 5

Min Row Index: 1 Max Row Index: 3
Min Column index: 1 Max Column index: 5

Min Row Index: 1 Max Row Index: 4
Min Column index: 1 Max Column index: 5

Min Row Index: 1 Max Row Index: 5
Min Column index: 1 Max Column index: 5

Min Row Index: 1 Max Row Index: 6
Min Column index: 1 Max Column index: 5
```

Output 5.4: ulbound.bas

The first Redim sets the initial bounds for the array. The additional Redims increase the number of rows in the array, while leaving the number of columns in intact. Keep in mind that arrays in FreeBasic are table-like, where each row has the same number of columns. Ragged arrays, where each row has a different number of columns, cannot be created in FreeBasic using arrays. You would have to use pointers to create a ragged array.

Array Initialization

You can initialize an array with values when using the Dim statement in a manner similar to initializing any of the other intrinsic data types, and type definitions. The following code snippet illustrates the syntax using a one dimensional array.

```
Dim aArray(1 to 5) As Integer => {1, 2, 3, 4, 5}
```

This code snippet dimensions a ubyte array with 5 elements, then sets the elements to the list contained within the curly brackets. The arrow operator, => tells the compiler that the list following the Dim statement should be used to initialize the array.

You can also dimension multidimensional arrays in the same manner, by specifying blocks of data enclosed within curly braces as the following code snippet illustrates.

```
Dim bArray(1 to 2, 1 to 5) As Integer => {{1, 2, 3, 4, 5}, _
                                         {6, 7, 8, 9, 10}}
```

In this example, the first block, {1, 2, 3, 4, 5}, corresponds to row 1, and the second block, {6, 7, 8, 9, 10}, corresponds to row 2. Remember that FreeBasic arrays are row-major, so the row is specified

before the column. When you initialize an array in this manner, you must be sure that the number of elements defined will fit into the array.

The following programs initializes two arrays and print the values to the console.

```
Dim aArray(1 To 5) As Integer => {1, 2, 3, 4, 5}
Dim bArray(1 To 2, 1 To 5) As Integer => {{1, 2, 3, 4, 5}, _
                                          {6, 7, 8, 9, 10}}
Dim As Integer i, j

'Print aArray values.
For i = 1 To 5
 Print "aArray(";i;" ):";aArray(i)
Next
Print

'Print bArray values.
For i = 1 To 2
For j = 1 To 5
Print "bArray(";i;",";j;" ):";bArray(i, j)
Next
Next


Sleep
End
```

Listing 5.5: arrayinit.bas

Analysis: Start by creating a single-dimension array and initializes the values of the array. Then create a two-dimensional array and initializes the both rows of the array. Using a For-Next loop we print the contents of the first array to the screen. The next For-Next loop prints the contents of the second to the screen. The program is then closed in the usual way.

When the program is ran, you should see the following output.

```
 aArray( 1 ): 1
 aArray( 2 ): 2
 aArray( 3 ): 3
 aArray( 4 ): 4
 aArray( 5 ): 5

 bArray( 1, 1 ): 1
 bArray( 1, 2 ): 2
 bArray( 1, 3 ): 3
```

```
bArray( 1, 4 ): 4
bArray( 1, 5 ): 5
bArray( 2, 1 ): 6
bArray( 2, 2 ): 7
bArray( 2, 3 ): 8
bArray( 2, 4 ): 9
bArray( 2, 5 ): 10
```

Output 5.5: arrayinit.bas

As you can see from the output, both arrays have been initialized correctly, and without using a loop to load the data into the array. You can initialize arrays of any size using this method, although large arrays get a bit hard to work with and it is important to maintain the proper order of data in the initialization block.

Using the -exx Compiler Switch

The -exx compiler switch will enable error and bounds checking within your program. If you go outside the bounds of an array within your program, the compiler will generate an “out of bounds” error while the program is running. This is a great help in debugging your program, and finding problems associated with arrays. -exx will also inform of you of Null pointer assignments, so it is quite useful when working with pointers as well. If you are using FBIde select View->Settings from the menu and the FreeBasic tab in the settings dialog. Add -exx to the end of your compiler command string.

Using -exx does add quite of bit of additional code to your program, so once your program. is functioning correctly, you will want to compile the program without the -exx switch.

Skill Test

1. How would you create an array of 10 integers?
2. How would you increase or decrease the number of integers in an array?
3. How do you find the upper and lower bound of an array?
4. What is the difference between a single and multi-dimensional array?
5. Create a single dimension array and set it’s values.
6. Create a multi-dimensional array and set it’s values.

Chapter 6 - Binary Number System

Introduction to Binary Numbers

Computers use the binary, or base 2, numbering system to represent data. Base 2 digits are the numbers 0 and 1. A single binary 1 or 0 is called a bit. Four bits is called a nybble. Two nybbles, or 8 bits is called a byte and 2 bytes make up a word. The size of a data type determines how many bits are available to represent a particular number. A byte has 8 bits, a short has 16 bits, an integer has 32 bits and a longint has 64 bits.

You will notice that each data type is double the size of the previous data type. This is because the binary system uses powers of 2 to represent data. 2^0 is 1. 2^1 is 2. 2^2 is 4. 2^3 is 8. 2^4 is 16, and so on. To find the value of a binary number, you start from the right, which is position 0 and add the power of twos going left if there is a 1 in the bit position. If a nybble is 1101, then the right-most position is 2^0, the next position left is skipped since it has a zero, followed by 2^2 and finally 2^3. Resolving the power terms gives you 1 + 4 + 8 which equals 13. The value ranges for the different data types is a direct result of the number of bits in each data type. Being able to manipulate individual bits, bytes and words has a number of uses. For example, the messaging system of the Windows API use integers to store both the id of a control and event the control received, as the following code snippet shows.

```
Case WM_COMMAND
wmId = Loword( wParam )
wmEvent = Hiword( wParam )
```

In this snippet the id of the control is stored in the low word of wParam, and the event number is stored in the high word. Since a word is 2 bytes or 16 bits, you can store65535 ids in a single word, using an unsigned data type, or 32767 ids for a signed datatype. This is a very efficient way to manage data in your program.

The Sign Bit

The sign bit, the leftmost bit, is used by the computer to determine if a signed datatype is negative or positive. To represent a negative number, the positive value of the number is negated. This is done changing all the 1's to 0 and the 0's to 1's, and 1 is added to that result. For example, binary 5 is 0000 0101. Negating all the bits results in 1111 1010. Adding 1 results in 11111011. Since the leftmost bit is 1, this is a negative number.

We can confirm this by using the power of 2 notation which results in the following:

$-128\ (2^7) + 64\ (2^6) + 32\ (2^5) + 16\ (2^4) + 8\ (2^3) + 0 + 2\ (2^1) + 1\ (2^0) = -5.$

Remember, if a bit is zero, we add zero to the total. The following program shows the binary representation of both positive 5 and negative five.

```
Dim As Byte myByte
Dim As String myBits

'Initialize variables
myByte = 5

'Get the binary form of number
myBits = Bin(myByte,8)

'Print out nybbles with a space between so is is
'easier to read
Print "myByte =";myByte;" which is binary ";
Print Mid(myBits, 1, 4) & " " & Mid(myBits, 5, 4)

myByte = -5
'Get the binary form of number
myBits = Bin(myByte,8)

'Print out nybbles with a space between so is is
'easier to read
Print "myByte =";myByte;" which is binary ";
Print Mid(myBits, 1, 4) & " " & Mid(myBits, 5, 4)

Sleep
End
```

Listing 6.1: signbit.bas

Analysis: At the start of the program we declare the working variables, a byte that will represent the actual number and a string that will represent the binary value. Next the myByte is set to 5. Afterwords the Bin function is used to return a string that represents the binary value of 5. Since Bin does not return any leading 0's, the String function is used to pad the string to a full 8 bits for display purposes.

The first parameter of the String function is the number of characters to add and the second parameter is the character to use to pad the string. Since a byte is 8 bits long, subtracting the length of myBits from 8 will give the number of 0's to add to the string, if the length of myBits is less than 8. The numeric value of myByte is then printed, which is 5. A semi-colon is added to the end of print statement so that Print will not print out a carriage return.

The binary string is printed in two groups of four, that is each nybble, to make the display easier to read. The Mid function used returns a portion of a string. The first parameter of the Mid function is the string, the second is the start position and the third parameter is the number of characters to copy. The first Mid returns the first four characters, which is appended to a space using the & operator, which in turn is appended to the last four characters. MyByte is set to -5 and the output is formatted and displayed it to the screen. The program is ended in the usual manner.

When you run the program the following output is displayed.

```
myByte = 5 which is binary 0000 0101
myByte =-5 which is binary 1111 1011
```

Output 6.1: Output of signbit.bas

You can see that the output for -5 matches the Two's Complement form. We can confirm this by negating 1111 1011 which results in 0000 0100 and adding 1 which results in 0000 0101, or positive 5.

Why is this important? Signed data types have a smaller range of values than unsigned data types because a bit is being used to represent the sign of the number. If you are needing to store a large number of data values, such as ids, in a byte or integer, the number of possible values you can store depends on whether it is signed or unsigned. If the number of values needed exceed the range of a signed data type, then use an unsigned data type.

Bit wise Operators

FreeBasic includes a number of operators that manipulate the bits of a number. The following table lists the bit wise operators in order of their precedence. That is, the first row has the highest precedence while lower rows have lower precedence.

Operator	Syntax	Truth Table	Comments
Not (Bit wise negation)	B = NOT expression	NOT 0 = 1 NOT 1 = 0	Inverts operand bit; turns a 1 into 0 and a 0 into 1.
And (Bit wise conjunction)	B = expression And expression	0 AND 0 = 0 1 AND 0 = 0 0 AND 1 = 0 1 AND 1 = 1	Result bit is 1 only if both operand bits are 1.
Or (Bit wise disjunction)	B = expression OR expression	0 OR 0 = 0 1 OR 0 = 1 0 OR 1 = 1 1 OR 1 = 1	Result bit is 1 if either or both operand bits is1.
Xor (Bit wise exclusion)	B = expression Xor expression	0 XOR 0 = 0 1 XOR 0 = 1	Result bit is 1 if operand bits are different.

Operator	Syntax	Truth Table	Comments
		0 XOR 1 = 1 1 XOR 1 = 0	
Eqv (Bit wise equivalence)	B = expression EQV expression	0 EQV 0 = 1 1 EQV 0 = 0 0 EQV 1 = 0 1 EQV 1 = 1	Result bit is 1 if both operand bits are 0 or 1.
Imp (Bit wise implication)	B = expression Imp expression	0 IMP 0 = 1 1 IMP 0 = 0 0 IMP 1 = 1 1 IMP 1 = 1	Result bit is 0 if first bit is 1 and second bit is 0, otherwise result is 1.

Table 6.1: Bit wise Operators

The truth table column indicates the operation on the individual bits. The order of the bits are not important except for the IMP operator which tests the bits of the second operand using the bits from the first operand.

The NOT Operator

You saw the NOT operator at work in the two's complement section. The following program performs the same two's complement operation.

```
Dim As Byte myByte = 5

'5 in decimal and binary
Print "myByte:";myByte," Binary: ";Bin(myByte)

'Apply NOT operator
myByte = Not myByte

'Value after NOT operation
Print "NOT myByte: ";myByte," Binary: ";Bin(myByte)

'Add 1 after NOT operation
myByte = myByte + 1

'Print result = 5 in decimal and binary
Print "myByte + 1: ";myByte," Binary: ";Bin(myByte)

Sleep
End
```

Listing 6.2: not.bas

Analysis: The program starts with the working variables being declared and Initialized. The Variable myByte is initialized to the value of 5. Next the value of myByte is printed. The NOT is performed on myByte and it is printed. One is added to myByte as required by the two's complement method. Then the decimal and binary values are printed out. The programs in the standard way.

When the program is run, you should see the following output.

```
myByte: 5      Binary: 101
NOT myByte: -6              Binary: 11111010
myByte + 1: -5              Binary: 11111011
```

Output 6.2: Output of not.bas

As you can see from the output, the final result of the program is -5 after applying the twos complement method to myByte. The 1 in the leftmost position indicates that the number is negative. Bin doesn't add the leading zeros in the first output line, but the three digits shown are the rightmost three digits.

The AND Operator

The AND operator can be used to test if an individual bit is 1 or 0 by using a mask value to test for the bit position as the following program illustrates.

```
'Declare working variable and mask value
'The mask will test the 3rd bit position, i.e. 4
Dim As Byte myByte = 5, Mask = 4

'Print decimal and binary values
Print "Testing 3rd bit position (from right)"
Print "myByte:";myByte," Binary: ";Bin(myByte)
Print "Mask: ";Mask," Binary: ";Bin(Mask)

'Check to see if 3rd bit is set
If (myByte And Mask) = 4 Then
 Print "3rd bit is 1"
Else
 Print "3rd bit is 0"
End If

Print

'The mask will test the 2nd bit position, i.e. 2
Mask = 2
```

```
'Print decimal and binary values
Print "Testing 2nd bit position (from right)"
Print "myByte:";myByte," Binary: ";Bin(myByte)
Print "Mask: ";Mask," Binary: ";Bin(Mask)

'Check to see if 2nd bit is set
If (myByte And Mask) = 4 Then
 Print "2nd bit is 1"
Else
 Print "2nd bit is 0"
End If
Sleep
END
```

Listing 6.3: and.bas

Analysis: At the start of the program, the working variables are declared and initialized. The first section of the program is testing for the third bit position of myByte, which is bit 2^2, or decimal 4. The next section prints out the heading, decimal and binary values for the variables. The If statement uses the AND operator to test for a 1 in the 3rd bit position, and since binary 5 contains a one in this position, the program will execute the code immediately following the Then keyword.

A mask value is set to 2 to test for a 1 in the second bit position, which is 2^1 or 2. The next section prints out the header, decimal and binary values of the variables. An AND operator is used to test for a 1, and since binary 5 has a 0 in this position, the program will execute the code immediately following the Else keyword. The program ends as normal.

Running the program produces the following output.

```
Testing 3rd bit position (from right)
myByte: 5      Binary: 101
Mask:  4       Binary: 100
3rd bit is 1

Testing 2nd bit position (from right)
myByte: 5      Binary: 101
Mask:  2       Binary: 10
2nd bit is 0
```

Listing 6.3: Output of and.bas

Looking at the binary values you can see how the bits line up and how the And operator can test for individual bits. 5 in binary has a bit set in the 2^0 (1) and 2^2 (4) position. Setting the mask value to 4 sets bit position 2^2 to 1 and all other bit positions to 0. The expression (myByte And Mask) will return an integer value that will contain the And values of the two operands. Since the mask has zeros in every

position except for the 2^2 position, all of the other bits will be masked out, that is 0, returning a 4. Since the return value of 4 matches the target value 4, the code following the Then clause is executed. The second portion of the program test for the 2^1 position of myByte. Since this position contains a 0 in myByte, the value returned from the expression (myByte And Mask) does not match the target value, so the code following the Else clause if executed. The program is then closed as normal.

The OR Operator

You can use the OR operator to set multiple values in a single variable. The Windows API uses this technique to set flags for objects such as the styles of a window.

The following program illustrates this concept.

```
'Declare working variable
Dim As Byte myByte, Mask

'Set the flags in the byte
myByte = 2
myByte = myByte Or 4

'Print decimal and binary values
Print "myByte set to 2 and 4"
Print "myByte:";myByte," Binary: ";Bin(myByte)
Print

'Set the mask to 2
mask = 2
Print "Testing for 2"

'Check for 2 value
If (myByte And Mask) = 2 Then
Print "myByte contains 2"
Else
 Print "myByte doesn't contains 2"
End If
Print

'Set the mask value to 4
Mask = 4

'Print decimal and binary values
Print "Testing for 4"
If (myByte And Mask) = 4 Then
 Print "myByte contains 4"
Else
 Print "myByte doesn't contain 4"
```

```
End If
Print

'Set the mask value to 8
Mask = 8

'Print decimal and binary values
Print "Testing for 8"
If (myByte And Mask) = 8 Then
 Print "myByte contains 8"
Else
Print "myByte doesn't contain 8"
End If

Sleep
End
```

Listing 6.4: or.bas

Analysis: The program starts by declaring the working variables. Next myByte is set to 2 and that value is combined with 4 using the OR operator. Then we print out the decimal and binary values of myByte. Next the mask is set to 2 and the AND operator is used to test for 2 in myByte. Since myByte contains a 2,the program will execute the code immediately following the Then clause. The same procedure is used to test for the value of 4. Since myByte contains a 4,the program will print out the text “myByte contains a 4”. The same procedure is used to test for 8, which myByte does not contains so the code after the THEN will be executed.

When you run the program, you should get the following output.

```
myByte set to 2 and 4
myByte: 6      Binary: 110

Testing for 2
myByte contains 2

Testing for 4
myByte contains 4

Testing for 8
myByte doesn't contain 8

myByte doesn't contain 8
```

Output 6.4: Output of or.bas

As you can see from the output, you can pack multiple values into a single variable. The number of values a variable can contain depends on the size of data type and the range of values. Using the OR

and AND combination is a technique that you will find in wide-spread use, especially in third-party libraries, as it provides an efficient way to pass multiple data items using a single variable.

The XOR Operator

One of the more useful aspects of the XOR operator is to flip bits between two states. XORing a value with 0 will return the original value, and XORing with a 1 returns the opposite value. Suppose that you start with a 1 bit and XOR with a 0 bit. Since the two inputs are different you will get a 1. If the start value is 0 and you XOR with a 0, then both values are the same and you will get a 0. In both cases the output is the same as the input. If you start with a 1 and XOR with a 1, you will get a 0 since both inputs are the same. If you start with a 0 and XOR with a 1, you will get a 1 since the inputs are different. Here the inputs have been flipped to the opposite values. You can use this technique with a mask value, XORing once to get a new value, and then XORing again with the same mask to get the original value.

One use of this technique is to display a sprite on the screen using XOR and then erasing the sprite by using another XOR at the same location. The first XOR combines the bits of the background with the sprite bits to display the image. Another XOR in the same location flips the bits back to their original state, once again showing the background and effectively erasing the sprite image.

XOR can also be used to swap the values of two variables as demonstrated in the following program.

```
Dim As Integer myInt1 = 5, myInt2 = 10
Print "myInt1 = ";myInt1;" myInt2 = ";myInt2
Print "Swapping values..."
myInt1 = myInt1 Xor myInt2
myInt2 = myInt1 Xor myInt2
myInt1 = myInt1 Xor myInt2
Print "myInt1 = ";myInt1;" myInt2 = ";myInt2
Sleep
End
```

Listing 6.6: xor.bas

Analysis: At the start of the program the working variables are declared and initialized. We then print out the initial variable values. Next we carry out the XOR operation. MyInt1 is then XORed with myInt2 to get an intermediate value which is stored in myInt1. The second XOR operation returns the original value of myInt1 which is stored in myInt2. The third XOR operation then returns the value of myInt2 from the intermediate value that was stored in myInt1. Next we print out the swapped values and the program is then closed in the usual way.

Running the program produces the following output.

```
myInt1 =  5 myInt2 =  10
Swapping values...
```

```
myInt1 =  10 myInt2 =  5
```

Output 6.5: Output of xor.bas

As you can see the program was able to swap the two values without using a temporary variable because the XOR operator is able to flip the bits between two distinct states.

The EQV Operator

The EQV operator isn't used much as a bit wise operator, but it can be used to see if two expressions are equivalent as the following program demonstrates.

```
#define False 0
#define True NOT False

Dim As Integer myInt1 = 4, myInt2 = 2

Print "myInt1 = ";myInt1;" myInt2 = ";myInt2
Print

'Both statements are true so are equivalent.
PRINT "Statement (myInt1 < 5) equiv (myInt2 > 1) ";
If (myInt1 < 5) Eqv (myInt2 > 1) = True Then
 Print "is equivalent."
Else
 Print "is not equivalent."
End If
Print

'Both statements are false so are equivalent.
Print "Statement (myInt1 > 5) equiv (myInt2 < 1) ";
If (myInt1 > 5) Eqv (myInt2 < 1) = True Then
 Print "is equivalent."
Else
 Print "is not equivalent."
End If
Print

'One is true, the other false so statement
'is not equivalent.
Print "Statement (myInt1 > 5) equiv (myInt2 < 1) ";
If (myInt1 > 3) Eqv (myInt2 < 1) = True Then
 Print "is equivalent."
Else
 Print "is not equivalent."
End If
```

```
SLEEP
End
```

Listing 6.7: eqv.bas

Analysis: The values for False and True are defined. FreeBasic uses -1 to indicate a True result from a logical operation such as that performed when executing an If statement. Since False is defined as 0, NOT False flips all the bits to 1, including the sign bit, making -1.

The working variables are declared and initialized. We then test the first expression. The If statement will execute the first expression, (myInt1< 5). Since 4 < 5 this will return True. The EQV operator has lower precedence then(myInt2 > 1), so this will be evaluated, and since 2 > 1 then will also return true. This leaves -1 EQV -1 to be evaluated. Since -1 is equivalent to -1, the whole statement is True. If Both of the expressions are False. 0 EQV 0 is True, so this statement is also True and will print out the affirmative. Next the first expression is True while the second is False. Since -1 is not equivalent to 0, this statement will evaluate to False and the negative will be printed.

When the program is run, you will see the following output.

```
myInt1 =  4 myInt2 =  2

Statement (myInt1 < 5) equiv (myInt2 > 1) is equivalent.

Statement (myInt1 > 5) equiv (myInt2 < 1) is equivalent.

Statement (myInt1 > 5) equiv (myInt2 < 1) is not equivalent.
```

Listing 6.7: Output of eqv.bas

It is important to realize that you are not testing to see if the expressions are True or False. You are only testing to see if the expressions are equivalent to each other. To put it another way, you are testing to see if two assertions are equivalent to each other. For example, suppose you have two characters in a game and you want to attack with both characters, if they are at equivalent strength. You could build an expression similar to the one in the listing and take action based on the equivalence of the two characters.

The IMP Operator

Like the EQV operator, IMP is rarely used as a bit wise operator. It is used in logic programming to test whether assertion A implies assertion B. Looking at the truth table we can see that a True assertion implies a True conclusion so True and True is also True. A True assertion however cannot imply a False conclusion so True and False is False. A False premise can imply any conclusion so False with any conclusion is always True. Unless you are building an expert system or natural language interface, you will probably never need to use this operator.

Caution Exercise caution when using bit wise operators with arithmetic operators, as the result may not be what you expect. When used in logical expressions, such as in an If statement, make sure the bit wise operators operate on either the True or False values of complete expressions to avoid evaluation problems.

Shortcut Bit wise Operators

The bit wise operators, like the arithmetic operators, can be written in shorthand form. The following table lists the shortcut versions of the bit wise operators.

Operator	Syntax	Comment
And=	B And= C	This is the same as B = B And C.
Or=	B Or= C	This is the same as B = B Or C.
Xor=	B Xor= C	This is the same as B = B Xor C.
Eqv=	B Eqv = C	This is the same as B = B Eqv C.
Imp=	B Imp= C	This is the same as B = B Imp C.

Table 6.2: Shortcut Bit wise Operators

The SHL and SHR Operators

SHL stands for shift left and SHR stands for shift right. As the names imply, these operators shift the bits of a byte or integer-type variable either left or right. The following table shows the syntax diagram of both operators.

Operator	Syntax	Comments
SHL (Shift bits left)	B = variable SHL number	Shifts the bits of variable left number of places.
SHR (Shift bits right)	B = variable SHR number	Shifts the bits of variable right number of places.

Table 6.3: SHL and SHR Operators

Shifting left is the same as multiplying by 2 and shifting right is the same as dividing by 2. You can see this by looking at the binary representation of a number. Take the byte value of 1 which is 0000 0001. The 1 is in the 2^0 position. 2^0 equals 1. Shifting the bit left, makes 0000 0010, putting the 1 bit in the 2^1 position, which evaluates to 2. This is the same as 1*2. Shifting left again puts the bit at the 2^2 position, 0000 0100which evaluates to 4, or 2 *2. Shifting the bit right puts the 1 back in the 2^1 position, 0000 0010, which evaluates to 2, or 4/2. Shifting right again puts the 1 bit in the 2^0 position, 0000 0001, which is 1 or 2/2.

The shift operation can be used as a replacement for multiplication or division if you are working with powers of 2, but it is primarily used to pack data items into variables, or to retrieve data items from variables. You saw this demonstrated in the MAKDWORD macro which was defined as #define MAKDWORD(x,y) (cint(x) shl 16 or cint(y)).

In this macro, the value of x is converted to an integer, and then shifted left 16 bits into the high word. An integer is 4 bytes and can be represented as 00000000 00000000 00000000 00000001 . Shifting 16 bits left makes 00000000 00000001 00000000 00000000. Remember that a word is two bytes, so the 1 has been shifted to the high word of the integer. You can then use the Hiword function to retrieve this value.

The following program shows the SHL and SHR operators.

```
‘Declare working variables
Dim As Uinteger myInt = 1, i

'Multiply by powers of 2
Print "Shifting left..."
Print "myInt = ";myInt

For i = 1 To 8
  myInt = myInt Shl 1
  Print "myInt = ";myInt
Next
Print

'Divide by powers of 2
Print "Shifting right..."
Print "myInt = ";myInt

For i = 1 To 8
 myInt = myInt Shr 1
 Print "myInt = ";myInt
Next

Sleep
End
```

Listing 6.8: shlr.bas

Analysis: The working variables are declared, myInt which will be the value that is shifted, and i for use in the For-Next loop The first is to shift myInt left 16 times, 1 bit position each time, and prints the result. The second loop then is used to shift the variable right16 times and prints the result. The program is closed in the usual way.

Running the program produces the following output.

```
Shifting left...
myInt = 1
myInt = 2
myInt = 4
myInt = 8
myInt = 16
myInt = 32
myInt = 64
myInt = 128
myInt = 256

Shifting right...
myInt = 256
myInt = 128
myInt = 64
myInt = 32
myInt = 16
myInt = 8
myInt = 4
myInt = 2
myInt = 1
```

Output 6.8: Output of shlr.bas

As you can see from the output, shifting left multiplies the value by 2 and shifting right divides the value by 2.

Bit wise Macros

FreeBasic has several built-in macros for retrieving and setting bit and byte data from a variable. The following tables lists the macros, the syntax and their definitions.

Macro	Syntax	Definition
Hiword	B = Hiword(variable)	#define Hiword(x) (CUInt(x) shr 16)
Loword	B = Loword(variable)	#define Loword(x) (CUInt(x) and 65535)
Hibyte	B = Hibyte(variable)	#define Hibyte(x) ((CUint(x) and 65280) shr 8)
Lobyte	B = Lobyte(variable)	#define Lobyte(x) (CUint(x) and 255)
Bit	B = Bit(variable, bit_number)	#define Bit(x, bit_number) (((x) and (1shl (bit_number))) > 0)

Macro	Syntax	Definition
Bitset	B = Bitset(variable, bit_number)	#define Bitset(x, bit_number) ((x) or (1shl (bit_number)))
Bitreset	B = Bitreset(variable, bit_number)	#define Bitreset(x, bit_number) ((x) and not (1 shl (bit_number)))

Table 6.4: Bit wise Macros

The Hiword macro returns the leftmost two bytes of an integer and the Loword macro returns the rightmost two bytes. The Hibyte macro returns the leftmost eight bits of an integer and the Lobyte returns the rightmost eight bits.

The Bit macro returns a -1 if a bit at position bit_number is a 1, otherwise it returns a 0. The Bitset macro sets the bit at position bit_number to 1 and returns the number, and the Bitreset macro sets the bit at position bit_number to 0 and returns the number. The rightmost bit is bit 0 following the binary numbering scheme.

Caution Bit wise operators will only work correctly on byte and integer-type data. A single or double-type variable that is passed to a bit wise operator will be implicitly converted to an integer, which may result in precision loss.

These macros are useful when working with third party libraries such as the Windows API, where several pieces of information are stored in a single data type.

Chapter 7 - Pointer Data Type

Introduction to Pointers

The pointer data type is unique among the FreeBasic numeric data types. Instead of containing data, like the other numeric types, a pointer contains the memory address of data. On a 32-bit system, the pointer data type is 4 bytes. FreeBasic uses pointers for a number of functions such as Image Create, and pointers are used heavily in external libraries such as the Windows API. Pointers are also quite fast, since the compiler can directly access the memory location that a pointer points to. A proper understanding of pointers is essential to effective programming in FreeBasic.

For many beginning programmers, pointers seem like a strange and mysterious beast. However, if you keep one rule in mind, you should not have any problems using pointers in your program. The rule is very simple: a pointer contains an address, not data. If you keep this simple rule in mind, you should have no problems using pointers.

Pointers and Memory

You can think of the memory in your computer as a set of post office boxes (P.O. Box) at your local post office. When you go in to rent a P.O. Box, the clerk will give you a number, such as 100. This is the address of your P.O. Box. You decide to write the number down on a slip of paper and put it in your wallet. The next day you go to the post office and pull out the slip of paper. You locate box 100 and look inside the box and find a nice stack of junk mail. Of course, you want to toss the junk mail, but there isn't a trash can handy, so you decide to just put the mail back in the box and toss it later. Working with pointers in FreeBasic is very similar to using a P.O. Box.

When you declare a pointer, it isn't pointing to anything which is analogous to the blank slip of paper. In order to use a pointer, it must be initialized to a memory address,which is the same as writing down the number 100 on the slip of paper. Once you have the address, find the right P.O. Box, you can dereference the pointer, open the mail box, to add or retrieve data from the pointed-to memory location. As you can see there are three basic steps to using pointers.

1. Declare a pointer variable.
2. Initialize the pointer to a memory address.
3. Dereference the pointer to manipulate the data at the pointed-to memory location.

This isn't really any different than using a standard variable, and you use pointers in much the same way as standard variables. The only real difference between the two is that in a standard variable, you can access the data directly, and with a pointer you must dereference the pointer to interact with the data. The following program illustrates the above steps.

```
 'Create a pointer doesn't point to anything.
Dim myPointer As Integer Ptr

'Initialize the pointer to point to 1 integer
myPointer = Callocate(1, Sizeof(Integer))

'Print the address
Print "Pointer address:";myPointer

'Add some meaningful data
*myPointer = 10

'Print the contents will not be garbage
Print "Memory location contains:";*myPointer

'Deallocate the pointer
Deallocate myPointer
Sleep
End
```

Listing 7.1: basicpointer.bas

Analysis: First a pointer to an integer is created. The pointer doesn't point to a memory location yet, and if you were to try and use this uninitialized pointer, you would generate an error. The Callocate function is used to set aside memory equal to the size of an integer, that is 4 bytes, and returns the starting address of the memory segment. The second parameter passed to Callocate is the size of the memory unit to allocate. The first parameter is the number of units to allocate. In this program, one unit of size integer is being allocated.

Now that myPointer has been initialized, you can use it. Next the contents of myPointer is printed, which illustrates that the variable contains a memory address and not data. Data is added to the memory location using the indirection operator *. The indirection operator tells the compiler you want to work with the data that myPointer is pointing to, rather than the memory address contained in the variable. Next the contents of the memory location, is printed, which is now 10. The memory allocated with Callocate is freed using the Deallocate procedure. The program is closed in the usual way.

Running the program should produce a result similar to the following output.

```
Pointer address:3089536
Memory location contains: 10
```

Output 7.1: basicpointer.bas

The address printed on your computer will probably be different, since the operating system allocates the memory used by Callocate.

When your program terminates, all memory that was allocated in the program is freed and returned to the operating system. However, it is good practice to deallocate memory when it is no longer needed, even if it isn't strictly necessary when a program terminates. The better you manage your program's memory, the less chance of problems you will have when running the program on different computer configurations.

Typed and Untyped Pointers

FreeBasic has two types of pointers, typed and untyped. The preceding program declared a typed pointer, `Dim myPointer as Integer Ptr`, which tells the compiler that this pointer will be used for integer data. Using typed pointers allows the compiler to do type checking to make sure that you are not using the wrong type of data with the pointer, and simplifies pointer arithmetic.

Untyped pointers are declared using the Any keyword:

```
Dim myPointer as Any Ptr
```

Untyped pointers have no type checking and default to size of byte. Untyped pointers are used in the C Runtime Library and many third party libraries, such as the Win32 API, to accommodate the void pointer type in C. You should use typed pointers so that the compiler can check the pointer assignments, unless working with libraries that require the void pointer.

Pointer Operators

There are two pointer operators in FreeBasic; the indirection operator and the address of operator.

Operator	Syntax	Comment
* (Indirection) [ ] (Index Access)	B = *myPointer *myPointer = B B = myPointer[index] myPointer[index] = B	You can access the data in a pointer memory location by either using the indirection operator or using index access. The index format uses the size of the data type to determine the proper indexing.
@ (AddressOf)	myPointer = @myVar myPointer = @mySub() myPointer = @myFunction()	Returns the memory address of a variable, subroutine or function.

Table 7.1: Pointer Operators

You will notice that the address of operator not only returns the memory address of a variable, but it can also return the address of a subroutine or function. You would use the address of a subroutine or

function to create a callback function such as used in the CRT function qsort. Callback functions will discussed later in this chapter.

Memory Functions

FreeBasic has a number of memory allocation functions that are used with pointers, as shown in the following table.

Function	Syntax	Comment
Allocate	myPointer = Allocate(number_of_bytes)	Allocates number_of_bytes and returns the memory address. If myPointer is 0, the memory could not be allocated. The allocated memory segment is not cleared and contains undefined data.
Callocate	myPointer = Callocate(number_of_elements, size_of_elements).	Callocate allocates number_of_elements that have size_of_elements and returns the memory address. If the memory could not be allocated, Callocate will return 0. The memory segment allocated is cleared
Deallocate	Deallocate myPointer	Frees the memory segment pointed to by myPointer.
Reallocate	myPointer = Reallocate(pointer, number_of_bytes)	Reallocate changes the sizeof a memory segment created with Allocate or Callocate. If the new size is larger than the existing memory segment, the contents of the memory segment remained unchanged. If the new size is smaller, the contents of the memory segment are truncated. If pointer is 0, Reallocate behaves just like Allocate. A 0 is returned if the memory segment cannot be changed.

Table 7.2: FreeBasic Memory Functions

These functions are useful for creating a number of dynamic structures such as linked lists, ragged or dynamic arrays and buffers used with third party libraries.

When using the Allocate function you must specify the storage size based on the data type using the equation number_of_elements * Sizeof(data type). To allocate space for 10 integers your code would look like this:

```
myPointer = Allocate(10 * Sizeof(Integer)).
```

An integer is 4 bytes so allocating 10 integers will set aside 40 bytes of memory. Allocate does not clear the memory segment, so any data in the segment will be random, meaningless data until it is initialized. Callocate works in the same fashion, except that the calculation is done internally. To allocate the same 10 integers using Callocate your code would look like this:

```
myPointer = Callocate(10, Sizeof(Integer))
```

Unlike Allocate, Callocate will clear the memory segment.

Reallocate will change the size of an existing memory segment, making it larger or smaller as needed. If the new segment is larger than the existing segment, then the data in the existing segment will be preserved. If the new segment is smaller than the existing segment, the data in the existing segment will be truncated. Reallocate does not clear the added memory or change any existing data.

All of these functions will return a memory address if successful. If the functions cannot allocate the memory segment, then a NULL pointer (0) is returned. You should check the return value each time you use these functions to be sure that the memory segment was successfully created. Trying to use a bad pointer will result in undesirable behavior or system crashes. There is no intrinsic method for determining the size of an allocation; you must keep track of this information yourself.

Caution: Be careful not to use the same pointer variable to allocate two or more memory segments. Reusing a pointer without first deallocating the segment it points to will result in the memory segment being lost causing a memory leak.

Pointer Arithmetic and Pointer Indexing

When you create a memory segment using the allocation functions, you will need away to access the data contained within the segment. In FreeBasic there are two methods for accessing data in the segment; using the indirection operator with pointer arithmetic, and pointer indexing.

Pointer arithmetic, as the name suggests, adds and subtracts values to a pointer to access individual elements within a memory segment. When you create a typed pointer such as Dim myPointer as Integer ptr, the compiler knows that the data being used with this pointer is of size Integer or 4 bytes. The pointer, when initialized, points to the first element of the segment. You can express this as *(myPtr + 0). To access the second element, you need to add 1 to the pointer, which can be expressed as *(myPtr + 1). Since the compiler knows that the pointer is an Integer pointer, adding 1 to the pointer reference will actually increment the address contained in myPtr by 4, the size of an Integer. This is why using typed pointers is preferable over untyped pointers. The compiler does much of the work for you in accessing the data in the memory segment.

Notice that the construct is *(myPtr + 1) and not *myPtr + 1. The * operator has higher precedence than +, so *myPtr + 1 will actually increment the contents myPtr points to, and not the pointer address. *myPtr will be evaluated first, which returns the contents of the memory location and then +1 will be evaluated, adding 1 to the memory location. By wrapping myPtr + 1 within parenthesis, you force the compiler to evaluate myPtr + 1 first, which increments the pointer address, and then the * is applied to return the contents of the new address.

Pointer indexing works the same way as pointer arithmetic, but the details are handled by the compiler. *(myPtr + 1) is equivalent to myPtr[1]. Again, since the compiler knows that myPtr is an integer pointer, it can calculate the correct memory offsets to return the proper values using the index. Which format you use if up to you, most programmers use the index method because of its simplicity.

The following program shows both methods of accessing a memory segment.

```
Dim myPtr As Integer Ptr
Dim As Integer i
'Try and allocate space for 10 integers
myPtr = Callocate(10, Sizeof(Integer))
'Make sure the space was allocated
If myPtr = 0 Then
 Print "Could not allocate space for buffer."
 End 1
End If
'Load data into the buffer
Print "Loading data, print data using *..."
For i = 0 To 9
 *(myPtr + i) = i
 Print "Index:";i;" data:";*(myPtr + i)
Next
Print
'Print data from buffer
Print "Show data using indexing..."
For i = 0 To 9
 Print "Index:";i;" data:";myPtr[i]
Next
'Free the memory
Deallocate myPtr
Sleep
End
```

Listing 7.2: ptraccess.bas

Analysis: The working variables are declared first. Space for 10 integers is created using the Callocate function. The next four lines check to make sure that the memory was allocated. If it wasn't, the program ends. The End 1 terminates the program with an exit code of 1. This is useful for

instances where the program may be run from a batch file and you want to make sure the program ran successfully. You can check the exit code in the batch file and take the appropriate action.

First the memory segment is printed using the indirection operator and pointer arithmetic. Next the same thing is done using the index method. Notice that the index method is much more compact and easier to read. The buffer is deallocated, even though it isn't strictly necessary as the program is terminating. Deallocating memory is a good habit to get into, even when it may not be strictly necessary. The program is closed in the usual way.

When you run the program you should see the following output.

```
Loading data, print data using *...
Index: 0 data: 0
Index: 1 data: 1
Index: 2 data: 2
Index: 3 data: 3
Index: 4 data: 4
Index: 5 data: 5
Index: 6 data: 6
Index: 7 data: 7
Index: 8 data: 8
Index: 9 data: 9


Show data using indexing...
Index: 0 data: 0
Index: 1 data: 1
Index: 2 data: 2
Index: 3 data: 3
Index: 4 data: 4
Index: 5 data: 5
Index: 6 data: 6
Index: 7 data: 7
Index: 8 data: 8
Index: 9 data: 9
```

Output 7.2: ptraccess.bas

As you can see from the output, both formats produce the same results, but the index method is a lot easier to read and understand, and less error-prone than the indirection method.

Pointer Functions

Freebasic has a set of pointer functions to complement the pointer operators. The following table lists the pointer functions.

Function	Syntax	Comment
Cptr	myPtr = Cptr(data_type, expression)	Converts expression to a data_type pointer. Expression can be another pointer or an integer.
Peek	B = Peek(data_type, pointer)	Peek returns the contents of memory location pointer to by pointer. Data_type specifies the type of expected data.
Poke	Poke data_type, pointer, expression	Puts the value of expression into the memory location pointed to by pointer. The data_type specifies the type of data being placed into the memory location.
Sadd	myPtr =Sadd(string_variable)	Returns the location in memory where the string data in a dynamic string is located.
Strptr	myPtr =Strptr(string_variable)	The same as Sadd.
Procptr	myPtr = Procptr(function)	Returns the address of a function. This works the same way as the address of operator @.
Varptr	myPtr = Varptr(variable)	This function works the same way as the address of operator @.

Table 7.3: Pointer Functions

The Sadd and Strptr functions will be discussed in the chapter on the string data types. The Peek and Poke functions have been added for the purposes of supporting legacy code. Procptr and Varptr both work just like the address of operator, but Proptr only works on subroutines and functions and Varptr only works on variables. Cptr is useful for casting an untyped pointer to a typed pointer, such as a return value from a third party library.

Subroutine and Function Pointers

Subroutines and functions, like variables, reside in memory and have an address associated with their entry point. You can use these addresses to create events in your programs, to create pseudo-objects and are used in callback functions. You create a sub or function pointer just like any other pointer except you declare your variable as a pointer to a subroutine or function rather than as a pointer to a data type. Before using a function pointer, it must be initialized to the address of a subroutine or function using Procptr or @. Once initialized, you use the pointer in the same manner as calling the original subroutine or function. The following program illustrates declaring an using a function pointer.

```
   'Declare our function to be used with pointer
```

```
   Declare Function Power(number As Integer, pwr As Integer) As
Integer

   'Dim a function pointer
   Dim FuncPtr As Function(x As Integer, y As Integer) As Integer

   'Get the address of the function
   FuncPtr = @Power

   'Use the function pointer
   Print "2 raised to the power of 4 is";FuncPtr(2, 4)

   Sleep
   End

   'Write the function that will be called

   Function Power(number As Integer, pwr As Integer) As Integer
    Return number^pwr
   End Function
```

Listing 7.3: funcptr.bas

Analysis: First the function prototype is declared, that will be used with the function pointer. Next the function pointer, FuncPtr, is declared using the As Function syntax. Notice that the Dim statement does not use the Ptr keyword; the compiler knows that this will be a function pointer since it is declared using the As Function method. When declaring a function pointer, the parameter list must match the number and type of parameters of the pointed-to function, but as you can see, the names do not have to match. In fact, the pointer can be declared as Dim FuncPtr As Function(As Integer, As Integer) As Integer, without the parameter names. The only requirement is to make sure that the type and number of parameters, and the return type, match the function declaration and definition. Next the function pointer is initialized to the address of the function using the address of operator @. You could use Procptr here as well. Now we call the function using the pointer to call the function. The calling syntax is the same as using the function name: FuncPtr(2,4) is equivalent to Power(2, 4). Lines 15 and 16 close the program in the usual way. The actual Power function is defined after the main part is closed.

Running the program will produce the following result.

```
   2 raised to the power of 4 is 16
```

Output 7.3: funcptr.bas

While this example program may not seem to have any advantages over just calling the function directly, you can use this method to call several functions using a single function pointer. For example,

if you were creating your own user interface, you could implement events using a function pointer that called one of several different subroutines depending on the object receiving the event. The only requirement would be that each subroutine must contain the same number and type of parameters.

Creating a Callback Function

One of the primary uses for function pointers is to create callback functions. A callback function is a function that you have created in your program that is called by a function or subroutine in an external library. Windows uses callback functions to enumerate through Window objects like fonts, printers and forms. The qsort, function contained within the C Runtime Library sorts the elements of an array using a callback function to determine the sort order. The prototype for the qsort function is contained instdlib.bi:

```
   declare sub qsort cdecl alias "qsort" (byval as any ptr, byval as
size_t, byval as size_t, byval as function cdecl(byval as any ptr,
byval as any ptr) as integer)
```

The following lists the parameter information for the qsort subroutine.

1. The first parameter is the address to the first element of the array. The easiest way to pass this information to qsort is to append the address of operator to the first element index: @myArray(0).
2. The second parameter is the number of elements in the array, that is the array count.
3. The third parameter is the size of each element in bytes. For an array of integers, the element size would be 4 bytes.
4. The fourth parameter is a function pointer to the user created compare function. The function must be declared using the Cdecl passing model, as shown in this parameter.

Using this information, you can see how qsort works. By passing the address of the first element along with the count of elements, and the size of each element, qsort can iterate through the array using pointer arithmetic. Qsort will take two array elements, pass them to your user defined compare function and use the compare function's return value to sort the array elements. It does this repeatedly until each array element is in sorted order. The following program uses the qsort subroutine and a compare function to sort an array of integers.

```
   #include "crt.bi"
   'Declare the compare function
   'This is defined in the same manner as the qsort declaration

   Declare Function QCompare Cdecl (Byval e1 As Any Ptr, Byval e2 As
Any Ptr) _
   As Integer

   'Dimension the array to sort
   Dim myArray(10) As Integer
```

```
Dim i As Integer

'Seed the random number generator
Randomize Timer

Print "Unsorted"

'Load the array with some random numbers
For i = 0 To 9
 'Rnd returns a number between 0 and 1
 'This converts the number to an integer
 myArray(i) = Int(Rnd * 20)

 'Print unsorted array
 Print "i = ";i;" value = ";myArray(i)
Next
Print

'Call the qsort subroutine
qsort @myArray(0), 10, Sizeof(Integer), @QCompare

Print

'Print sorted array.
Print "Sorted"

For i = 0 To 9
 'Rnd returns a number between 0 and 1 to convert to integer
 Print "i = ";i;" value = ";myArray(i)
Next

Sleep
End


'The qsort function expects three numbers
'from the compare function:
'-1: if e1 is less than e2
'0: if e1 is equal to e2
'1: if e1 is greater than e2

Function QCompare Cdecl (Byval e1 As Any Ptr, _
                         Byval e2 As Any Ptr) As Integer
 Dim As Integer el1, el2
 Static cnt As Integer
```

```
    'Get the call count and items passed
    cnt += 1
    'Get the values, must cast to integer ptr
    el1 = *(Cptr(Integer Ptr, e1))
    el2 = *(Cptr(Integer Ptr, e2))

    Print "Qsort called";cnt;" time(s) with";el1;" and";el2;"."

    'Compare the values
    If el1 < el2 Then
         Return 1
    Elseif el1 > el2 Then
         Return 1
    Else
         Return 0
    End If
  End Function
```

Listing 7.4: qsort.bas

Analysis: The crt.bi file is included so that the qsort routine will be available in the program. You need to include this file if you want to use any of the CRT functions. Next we declare the compare function. You will notice that it is declared as a Cdecl function, which matches the 4th parameter definition in the qsort declaration. Since qsort expects to see either a -1, 0 or 1, the function's return type is an integer.

When you see a line that ends with the underscores character, that is the line continuation character. The line continuation character and tells the compiler that the following line is actually a part of this line. Next an array of integers is dimensioned. The array will have 10 elements, with indexes from 0 to 9. dimensions a working variable i, that will be used to load and display the array values.

Next we seed the random number generator by using the value from the Timer function. The Timer function returns the number of seconds, as a double-type value, since the computer was started. A For-Next loop initializes the array with some random integers and displays them to the console screen. The code , Int(Rnd * 20), uses the Rnd function which returns a double precision number between 0 and 1. That number is multiplied by twenty to produce a double-precision number between 0and 20 which is then converted to an integer value using the Int function. Since Int returns the largest integer less than or equal to the input, the resulting random number swill range from 0 to 19. Then the value of the current array index is printed to the console screen.

The qsort subroutine is called. Since the array indexes range from 0 to 9, the address of operator on the zero index in used for the first parameter. The array size is10, so that is used as the second parameter. The array is an integer array, so Sizeof(Integer) is used to pass the array element size. The final parameter is the address of the compare function which is passed to qsort using the address of

operator. A For-next loop then prints out the now sorted array. The program is then closed in the usual way.

The next block of code contains the compare function. The function is defined just like the declare statement at the start of the program, using two Any Ptr parameters and returning an integer. The Any Ptrs allow qsort to be able to sort any type of data, including composite types. Next we delcare the function's working variables. Qsort will pass pointers to the functions, not the actual data, so two integers need to be declared so that the data that the pointers point to can be compared. The variable cnt is defined as Static so that its value will be preserved between calls to the function. Cnt is used to keep track of how many calls to the function qsort will make as it sorts the array. The indirection operator to return the data from the passed pointers. Notice the Cptr is used to convert the Any Ptr parameters to Integer Ptrs before using the indirection operator. Remember that the code inside the parenthesis will be executed before applying the indirection operator. We then print out the current call count and the passed parameter values.

We now compare the two values and return the appropriate indicator value back to qsort. The Return statement, as expected, is used to set the function's return value. The function is closed using the End Function keywords.

When you run the program you should something similar to the following.

```
Unsorted
i =  0 value =  17
i =  1 value =  19
i =  2 value =  5
i =  3 value =  7
i =  4 value =  5
i =  5 value =  6
i =  6 value =  6
i =  7 value =  14
i =  8 value =  15
i =  9 value =  14

Qsort called 1 time(s) with 17 and 6.
Qsort called 2 time(s) with 6 and 14.
Qsort called 3 time(s) with 17 and 6.
Qsort called 4 time(s) with 19 and 6.
Qsort called 5 time(s) with 15 and 6.
Qsort called 6 time(s) with 14 and 6.
Qsort called 7 time(s) with 6 and 6.
Qsort called 8 time(s) with 5 and 6.
Qsort called 9 time(s) with 7 and 6.
Qsort called 10 time(s) with 5 and 6.
Qsort called 11 time(s) with 7 and 6.
Qsort called 12 time(s) with 6 and 6.
```

```
   Qsort called 13 time(s) with 5 and 7.
   Qsort called 14 time(s) with 5 and 5.
   Qsort called 15 time(s) with 19 and 5.
   Qsort called 16 time(s) with 14 and 19.
   Qsort called 17 time(s) with 15 and 14.
   Qsort called 18 time(s) with 17 and 15.
   Qsort called 19 time(s) with 5 and 7.
   Qsort called 20 time(s) with 5 and 5.
   Qsort called 21 time(s) with 19 and 5.
   Qsort called 22 time(s) with 14 and 19.
   Qsort called 23 time(s) with 15 and 14.
   Qsort called 24 time(s) with 5 and 7.
   Qsort called 25 time(s) with 5 and 5.
   Qsort called 26 time(s) with 19 and 5.
   Qsort called 27 time(s) with 14 and 19.
   Qsort called 28 time(s) with 5 and 7.
   Qsort called 29 time(s) with 5 and 5.
   Qsort called 30 time(s) with 19 and 5.
   Qsort called 31 time(s) with 5 and 7.
   Qsort called 32 time(s) with 5 and 5.
   Qsort called 33 time(s) with 5 and 7.

   Sorted
   i =  0 value =  14
   i =  1 value =  6
   i =  2 value =  6
   i =  3 value =  7
   i =  4 value =  5
   i =  5 value =  5
   i =  6 value =  19
   i =  7 value =  14
   i =  8 value =  15
   i =  9 value =  17
```

Output 7.4: Output of qsort.bas

The first group of numbers show the unsorted array. The middle group of numbers show the number of times the compare function is called along with the values being sorted. The last group of numbers show the sorted array. Even though qsort is called quite a number of times even on this small array, the routine is extremely fast since it uses pointers to sort the values in the array.

Pointer to Pointer

In FreeBasic you can create a pointer to any of the supported data types, including the pointer data type. A pointer to a pointer is useful in situations where you need to return a pointer to a function or in

creating specialized data structures such as linked-lists and ragged arrays. A pointer to a pointer is called multi-level indirection.

Caution You can have as many levels of indirection as needed, but anything beyond two levels is rarely useful and difficult to manage.

One application of a pointer to pointer is the creation of memory arrays. The following program demonstrates the creation, manipulation and freeing of a memory
array.

```
'Declare a pointer to an int pointer

Dim myMemArray As Integer Ptr Ptr
Dim As Integer i, j

'Create 10 rows of integer pointers
myMemArray = Callocate(10, Sizeof(Integer Ptr))

'Add 10 columns of integers to each row
For i = 0 To 9
 myMemArray[i] = Callocate(10, Sizeof(Integer))

Next

'Add some data to the memory segment
For i = 0 To 9
 For j = 0 To 9
      myMemArray[i][j] = Int(Rnd * 10)
 Next
Next

'Print out data
For i = 0 To 9
 For j = 0 To 9
      Print "i,j = ";i;",";j;" Mem Array =";myMemArray[i][j]
 Next
Next

'Free memory segment
For i = 0 To 9
 Deallocate myMemArray[i]
Next

'Free the pointer to pointer
DeAllocate myMemArray
Sleep
```

```
End
```

Listing 7.5: memarray.bas

Analysis: We start by declaring a pointer to an integer pointer, myMemArray, which will simulate a two dimensional array, with myMemArray pointing to a list of integer pointers. This list of pointers will comprise the “rows” of the array. Next we just declares some working variables that are used in the For-Next loops later in the program.

The we create the rows of the array by allocating a memory segment that will contain 4 integer pointers, which are initialized in lines . Remember that the index method of accessing a pointer automatically does the pointer arithmetic for you. The code iterates through the memory segment and initializes the memory segment with pointers to memory segments that will contain integers. In other words, you have a pointer that is pointing to a list of pointers. These newly created memory segments are the columns for each row. Each column index will be a pointer to a memory location containing an integer.

Some random numbers are added to the memory array. Notice that by using the indexing method you can access the memory array just like a normal array. ipoints to the row (the list of pointers) and j points to the individual columns within that row which contain the integer data. The same method is used to print out the array values.

We then free the individual rows of memory. It is important that each row be freed before freeing myMemArray. If you were to just freemy MemArray, the rows would still be in memory, but unaccessible, causing a memory leak. Once all the rows have been freed, myMemArray can be freed. Since the program is terminating, Deallocating the memory is not strictly required in this instance, but if you needed to reuse the memory, then you must Deallocate in the method described, otherwise you will get a memory leak while the program is running. The program is then closed in the usual way.

Running the program should produce a result similar to the following.

```
i,j = 0, 1 Mem Array = 5
i,j = 0, 2 Mem Array = 1
i,j = 0, 3 Mem Array = 8
i,j = 0, 4 Mem Array = 5
i,j = 1, 0 Mem Array = 4
i,j = 1, 1 Mem Array = 3
i,j = 1, 2 Mem Array = 8
i,j = 1, 3 Mem Array = 8
i,j = 1, 4 Mem Array = 7
i,j = 2, 0 Mem Array = 1
i,j = 2, 1 Mem Array = 8
i,j = 2, 2 Mem Array = 7
i,j = 2, 3 Mem Array = 5
```

```
i,j = 2, 4 Mem Array = 3
i,j = 3, 0 Mem Array = 0
i,j = 3, 1 Mem Array = 0
i,j = 3, 2 Mem Array = 3
i,j = 3, 3 Mem Array = 1
i,j = 3, 4 Mem Array = 1
i,j = 4, 0 Mem Array = 9
i,j = 4, 1 Mem Array = 4
i,j = 4, 2 Mem Array = 1
i,j = 4, 3 Mem Array = 0
i,j = 4, 4 Mem Array = 0
```

Output 7.5: Output of memarray.bas

As you can see from the output, the memory array behaves exactly like a predefined array. This structure is useful for adding dynamic arrays to type definitions, which normally cannot hold a dynamic array. You will see this in more detail in the chapter on composite types.

One last note on this program. If you run the program more than once, you will notice that the values in the array are always the same, even though the program is generating random numbers. This is because the program did not seed the random number generator using the Randomize statement. To get different numbers for each run, add Randomize Timer before calling the Rnd function.